Математические алгоритмы для программистов

3D-графика, машинное обучение и моделирование на Python

Пол Орланд



Math for Programmers

3D graphics, machine learning and simulations with Python

PAUL ORLAND



Shelter Island

Математические алгоритмы для программистов

3D-графика, машинное обучение и моделирование на Python

Пол Орланд



Санкт-Петербург • Москва • Минск

2023

ББК 32.973.2-018
УДК 004.42
О-66

Орланд Пол

О-66 Математические алгоритмы для программистов. 3D-графика, машинное обучение и моделирование на Python . — СПб.: Питер, 2023. — 752 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2287-5

Неважно, чем вы занимаетесь — большими данными, машинным обучением, компьютерной графикой или криптографией, — без математики вам не обойтись! Везде сейчас требуются базовые знания и понимание алгоритмов.

Практические примеры позволят легко разобраться с самыми необходимыми математическими понятиями. 300 упражнений и мини-проектов откроют вам новые возможности в освоении интересных и популярных IT-профессий. Вы познакомитесь с базовыми библиотеками Python, используемыми при разработке реальных приложений, и вспомните давно забытые основы линейной алгебры и матана.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018
УДК 004.42

ISBN 978-1617295355 англ.
ISBN 978-5-4461-2287-5

Краткое содержание

Часть III
Машинное обучение

Оглавление

Часть I
Векторы и графика

Часть II
Математический анализ и моделирование физического мира

Часть III
Машинное обучение



Посвящается папе — моему первому учителю математики и информатики.

Предисловие

Я начал работать над этой книгой в 2017 году, будучи техническим директором организации Tachyus, которую я основал. В ней мы разрабатываем программное обеспечение для прогностической аналитики в нефтегазоперерабатывающей промышленности. К тому времени мы закончили работу над нашим основным продуктом — симулятором движения жидкостей, который был создан в том числе с использованием машинного обучения. Этот инструмент позволил нашим клиентам заглянуть в будущее своих нефтяных месторождений и выявить новые возможности экономии на сотни миллионов долларов.

Моя задача как технического директора состояла в масштабировании разработанного программного обеспечения по мере ввода в эксплуатацию в некоторых крупнейших компаниях мира. Трудность была не только в высокой сложности проекта, но и в том, что код был переполнен математическими вычислениями. Примерно в то же время мы начали искать сотрудников на должность, которая называлась «научный инженер-программист», полагая, что нам нужны высококвалифицированные инженеры-программисты, хорошо владеющие математикой, физикой и машинным обучением. В ходе поиска и найма специалистов я понял, что такое сочетание профессиональных навыков встречается редко и пользуется большим спросом. Наши инженеры-программисты тоже осознали это и стремились отточить свои знания математики, чтобы внести вклад в разработку специализированных серверных компонентов, составляющих наш стек. Поскольку в нашей команде уже были специалисты, с энтузиазмом изучающие математику, то в процессе найма я начал задумываться о том, как же из сильного инженера-программиста сделать еще и отличного математика.

К своему разочарованию, я обнаружил, что на рынке нет достойных книг по математике для программистов. Конечно, есть сотни книг и тысячи бесплатных онлайн-статей на такие темы, как линейная алгебра и математический анализ, но не было ничего, что я мог бы дать обычному инженеру-программисту и быть уверенным, что тот освоит необходимое за несколько месяцев. Я говорю это

не для того, чтобы принизить инженеров-программистов, я просто отмечаю, что читать книги по математике и разбираться в них — это сложный навык, которому трудно научиться самостоятельно. Для этого сначала нужно понять, какие конкретные темы вам следует изучить (что довольно сложно, особенно если не знать, что в действительности нужно!), прочитать книги по этим темам, а затем подобрать подходящие упражнения и попрактиковаться в применении новых знаний. Иначе можно прочитать учебник от корки до корки, решить *все* упражнения, предложенные в нем, потратив кучу времени!

В своей книге «Математические алгоритмы для программистов» я надеялся организовать альтернативную возможность. Я считаю, что можно прочитать эту книгу от первого до последнего слова и выполнить все упражнения за разумное время, а затем взяться за работу, уже владея некоторыми ключевыми математическими концепциями.

КАК СОЗДАВАЛАСЬ ЭТА КНИГА

Осенью 2017 года я связался с издательством Manning, рассказал им о своей задумке, и они подтвердили, что были бы заинтересованы в издании такой книги. Это положило начало долгому процессу преобразования моего видения книги в конкретный план, что оказалось намного сложнее, чем я себе представлял, будучи начинающим автором. В издательстве мне задали несколько трудных вопросов по содержимому будущей книги.

- Будет ли кому-то интересна эта тема?
- Не будет ли она чересчур абстрактной?
- Вы действительно сможете вместить семестровую программу преподавания численных методов в одну главу?

Все эти вопросы заставили меня тщательнее оценить достижимость цели. Я расскажу, как мы ответили на некоторые из них, чтобы вы могли понять, как устроена эта книга.

Во-первых, я решил раскрыть в книге один из ключевых приемов — выражение математических идей в коде. Я думаю, что это отличный способ обучения математике, даже если вы не программист по профессии. Когда я учился в старших классах, я научился программировать на графическом калькуляторе TI-84. У меня возникла грандиозная идея написать программы, которые будут делать за меня домашнюю работу по математике и естественным наукам, давая правильный ответ *и* раскладывая решение по шагам. Как и следовало ожидать, это оказалось намного сложнее, чем просто выполнять домашние задания, зато я получил полезный опыт. В любой задаче, которую я хотел запрограммировать, мне нужно было четко понимать, что имеется

на входе и что должно получиться на выходе, а также то, что происходит на каждом шаге решения. В результате я получил надежные знания и рабочую программу, подтверждающую это.

Этим опытом я и постараюсь поделиться с вами в книге. Каждая глава основана на реальном примере программы, для запуска которой нужно правильно собрать воедино все математические компоненты. Сделав это, вы будете уверены, что поняли идею и сможете применить ее снова в будущем. Я включил в текст множество упражнений, чтобы вы могли проверить, что действительно разобрались в математике и представленном программном коде, а также мини-проекты, предлагающие поэкспериментировать.

Еще один вопрос, который я обсуждал с Manning, заключался в выборе языка программирования для примеров. Первоначально я хотел использовать функциональный язык программирования, потому что математика сама является функциональным языком. В конце концов, понятие «функции» возникло в математике задолго до того, как появились компьютеры. В различных областях математики есть функции, которые возвращают другие функции, например интегралы и производные. Однако, если попросить читателей выучить незнакомый язык, такой как LISP, Haskell или F#, *одновременно* с изучением новых математических понятий, это серьезно усложнит книгу. Поэтому было решено остановиться на Python — популярном и простом в освоении языке с отличными математическими библиотеками. Python также фаворит для пользователей из реального мира — как ученых, так и разработчиков.

Последний важный вопрос, на который мне нужно было ответить, состоял в том, какие именно математические темы я бы включил в книгу, а какие опустил. Это было трудное решение, но по крайней мере мы договорились о названии «Математические алгоритмы для программистов», широта которого давала некоторую гибкость в выборе содержания книги. Вот мой главный критерий: это должны быть «Математические алгоритмы для программистов», а не «Математические алгоритмы для ИТ-специалистов». С учетом этого я мог бы опустить такие темы, как дискретная математика, комбинаторика, графы, логика, нотация «О большое» и т. д., которые изучаются на уроках информатики и в основном используются для *изучения* программ.

Но даже после того как это решение было принято, на выбор оставался широкий круг математических тем. В конце концов я решил сосредоточиться на линейной алгебре и численных методах. У меня есть определенные педагогические взгляды на эти темы и много хороших примеров визуальных и интерактивных приложений. Вы можете написать большой учебник *либо только* по линейной алгебре, *либо* по численным методам и тем самым еще больше сузить круг тем. Я решил, что книга будет основываться на некоторых приложениях в модной нынче области машинного обучения. После принятия этих решений содержание книги стало вырисовываться яснее.

ОХВАТЫВАЕМЫЕ МАТЕМАТИЧЕСКИЕ ИДЕИ

Эта книга охватывает множество математических тем, но основных — несколько. Вот некоторые из них.

- *Многомерные пространства.* Вы наверняка понимаете, что означают слова «двухмерное пространство» и «трехмерное пространство». Мы с вами живем в трехмерном мире и можем представить двухмерный мир как плоский лист бумаги или экран компьютера. Местоположение в двухмерном пространстве можно описать двумя числами (их часто называют координатами x и y), а местоположение в трехмерном пространстве определяется тремя числами. Мы не можем вообразить 17-мерное пространство, но можем описать его точки списками из 17 чисел. Такие списки чисел называются *векторами*, а векторная математика помогает прояснить понятие «-мерности».
- *Пространства функций.* Иногда список чисел может задавать функцию. Например, два числа, такие как $a = 5$ и $b = 13$, могут задавать (линейную) функцию вида $f(x) = ax + b$, и в этом случае функция будет иметь конкретный вид $f(x) = 5x + 13$. Для каждой точки в двухмерном пространстве с координатами (a, b) существует связанная с ней линейная функция. Таким образом, множество всех линейных функций можно представить как двухмерное пространство.
- *Производные и градиенты.* Это вычислительные операции, измеряющие скорость изменения функций. *Производная* говорит о том, как быстро функция $f(x)$ увеличивается или уменьшается с увеличением входного значения x. Функцию в двухмерном пространстве можно представить как $f(x, y)$, и она может увеличиваться или уменьшаться при изменении значений x или y. Если представить пары (x, y) в виде точек в двухмерном пространстве, может возникнуть вопрос: в каком направлении следует двигаться в этом пространстве, чтобы функция f увеличивалась быстрее всего? Ответ на этот вопрос дает градиент.
- *Оптимизация функции.* Для функции вида $f(x)$ или $f(x, y)$ можно сформулировать еще более широкую версию предыдущего вопроса: какие входные данные дают наибольшее значение функции? Для $f(x)$ ответом будет некоторое значение x, а для $f(x, y)$ — точка в двухмерном пространстве. В случае двухмерного пространства нам может помочь градиент. Если он говорит, что $f(x, y)$ увеличивается в каком-то направлении, то можно найти максимальное значение $f(x, y)$, следуя в этом направлении. Аналогичная стратегия применяется для поиска минимального значения функции.
- *Прогнозирование данных с помощью функций.* Допустим, вы хотите предсказать число, например цену акции в определенный момент времени. Можете создать функцию $p(t)$, которая принимает время t и выводит цену p. Мерой прогностического качества такой функции является близость к фактическим данным. В этом смысле поиск прогностической функции сводится к мини-

мизации ошибки между вашей функцией и реальными данными. Для этого нужно исследовать пространство функций и найти минимальное значение. Это называется *регрессией*.

Я думаю, что знание этих математических понятий пригодится каждому, даже тем, кто не интересуется машинным обучением, потому что эти и другие концепции, представленные в книге, имеют множество других применений.

Темы, которые мне скрепя сердце пришлось оставить за рамками книги, — это вероятность и статистика. Вероятность как концепция количественной оценки неопределенности в целом играет важную роль в машинном обучении. Но эта книга и без того получилась довольно объемной, поэтому мне просто не хватило бы ни времени, ни места, чтобы втиснуть хоть сколько-нибудь достойное внимания введение в эти темы. Следите за продолжением книги. Помимо тем, которые я смог охватить на этих страницах, есть еще много интересных и полезных сведений, и я надеюсь, что смогу поделиться ими с вами в будущем.

Благодарности

Я работал над этой книгой около трех лет. За эти годы я получил помощь от многих людей, и поэтому у меня очень длинный список тех, кого я хотел бы поблагодарить.

Прежде всего, я благодарен издательству Manning за помощь в подготовке этой книги. Они поверили, что я, начинающий автор, смогу написать эту большую и сложную книгу, и проявили большое терпение, так как работа над ней не раз отставала от графика. В частности, хочу сказать спасибо Марьян Бейс (Marjan Bace) и Майклу Стивенсу (Michael Stephens) за продвижение проекта и за то, что помогли определить, каким он должен быть. Мой первоначальный редактор-консультант Ричард Уоттенбергер (Richard Wattenbarger) тоже сыграл ключевую роль в создании книги, помогая снова и снова уточнять содержание. Он просмотрел шесть черновых вариантов глав 1 и 2, прежде чем мы решили, как будет структурировано издание.

Большую часть книги я написал в 2019 году под чутким руководством моего второго редактора Дженнифер Стаут (Jennifer Stout), которая довела проект до финишной черты и многому научила меня в области написания технических книг. Научный редактор Крис Ати (Kris Athi) и рецензент Майк Шепард (Mike Shepard) прошли с нами весь путь. Они внимательно вычитали каждое слово и каждую строку кода и исправили бесчисленное количество ошибок. Немалую помощь в работе мне оказала Микаэла Леунг (Michaela Leung), которая проверила всю книгу на предмет грамотности и технической точности. Я также хотел бы поблагодарить маркетинговую команду Manning. Благодаря программе раннего доступа к книгам MEAP (Manning Early Access Program) мы смогли убедиться, что эта книга будет интересна людям. Знание того, что она будет иметь хотя бы скромный коммерческий успех, стало отличным мотиватором на заключительных напряженных этапах ее подготовки к публикации.

Мои нынешние и бывшие коллеги из Tachyus многому научили меня в программировании, и часть их уроков вошли в эту книгу. Я благодарю Джека Фокса (Jack Fox) за то, что он первым заставил меня задуматься о связи между

функциональным программированием и математикой, о которой рассказывается в главах 4 и 5. Уилл Смит (Will Smith) учил меня проектированию видеоигр, и у нас состоялось множество содержательных дискуссий о векторной геометрии для отображения трехмерных сцен. Стелиос Кириаку (Stelios Kyriacou) научил меня почти всему, что я знаю об алгоритмах оптимизации, и помог отладить часть кода из книги. Он также познакомил меня с философией, согласно которой «любая задача является задачей оптимизации»— ее обсуждением мы займемся во второй половине книги.

Спасибо всем рецензентам: Адиру Рамджиавану (Adhir Ramjiawan), Анто Аравинту (Anto Aravinth), Кристоферу Хаупту (Christopher Haupt), Клайву Харберу (Clive Harber), Дэну Шейху (Dan Sheikh), Дэвиду Онгу (David Ong), Дэвиду Тримму (David Trimm), Эмануэлю Пиччинелли (Emanuele Piccinelli), Федерико Бертолуччи (Federico Bertolucci), Фрэнсису Буонтемпо (Frances Buontempo), Герману Гонсалесу-Моррису (German Gonzalez-Morris), Джеймсу Ньике (James Nyika), Йенсу Кристиану Б. Медсену (Jens Christian B. Madsen), Йоханнесу ван Неймегену (Johannes Van Nimwegen), Джонни Хопкинсу (Johnny Hopkins), Джошуа Хорвицу (Joshua Horwitz), Хуану Руфесу (Juan Rufes), Кеннету Фрикласу (Kenneth Fricklas), Лоуренсу Джиглио (Laurence Giglio), Натану Мише (Nathan Mische), Филипу Бесту (Philip Best), Река Хорвату (Reka Horvath), Роберту Уолшу (Robert Walsh), Себастьену Портебуа (Sébastien Portebois), Стефано Палуэлло (Stefano Paluello) и Винсенту Жу (Vincent Zhu). Ваши замечания и предложения помогли сделать эту книгу лучше.

Я не эксперт по машинному обучению, поэтому обращался к множеству источников, чтобы убедиться, что представил его точно и правильно. Больше всего на меня повлиял курс Эндрю Ына (Andrew Ng) *Machine Learning* на Coursera и серия *Deep Learning* от 3Blue1Brown на YouTube. Это отличные ресурсы, и если вы обращались к ним, то заметите, как они повлияли на подачу сведений в трети книги. Я также должен поблагодарить Дэна Рэтбоуна (Dan Rathbone), чей веб-сайт CarGraph.com был источником данных для многих моих примеров.

Я также хочу сказать спасибо своей супруге Маргарет, астроному, за то что познакомила меня с блокнотами Jupyter. Перенос примеров кода для этой книги в блокноты Jupyter значительно упростил их опробование. Мои родители тоже оказали значительную поддержку, когда я писал книгу: несколько раз случалось так, что я пытался закончить очередную главу во время праздничного визита к ним. Еще они лично гарантировали, что я обязательно продам хотя бы один экземпляр (спасибо, мама!).

Наконец, эта книга посвящена моему папе, который первым показал мне, как реализуется математика в программном коде, обучая меня программированию на APL, когда я учился в пятом классе. Если выйдет второе издание, я мог бы заручиться его помощью, чтобы переписать весь код на Python одной строкой на APL!

Об этой книге

Книга научит вас решать математические задачи с помощью программного кода на языке Python. Математические навыки становятся все более важными для профессиональных разработчиков программного обеспечения, особенно в связи с тем, что компании комплектуют команды, занимающиеся исследованием данных и машинным обучением. Математика играет важнейшую роль и в других областях, таких как разработка игр, компьютерная графика и анимация, обработка изображений и сигналов, система ценообразования и анализ фондового рынка.

Мы начнем с введения в двух- и трехмерную векторную геометрию, векторные пространства, линейные преобразования и матрицы, составляющие основу линейной алгебры. В части II будут представлены численные методы с упором на наиболее полезные для программистов темы: производные, градиенты, метод Эйлера и символьные вычисления. Наконец, в части III все перечисленное ранее собирается вместе, чтобы показать, как работают некоторые важные алгоритмы машинного обучения. К последней главе книги вы будете знать математику на достаточном уровне, чтобы написать собственную нейронную сеть с нуля.

Но имейте в виду, это не учебник! Цель книги — дать адаптированное введение в темы, которые могут показаться пугающими, запутанными или скучными. В каждой главе демонстрируется пример практического применения математической концепции, дополненный упражнениями, которые помогут проверить, как вы поняли материал, а также мини-проектами, которые помогут продолжить исследование.

КОМУ АДРЕСОВАНА КНИГА

Издание адресовано всем, кто имеет значительный опыт программирования и хочет освежить свои знания по математике или узнать больше о ее применении в программном обеспечении. Для этого не требуется разбираться в матанализе или линейной алгебре — достаточно знания алгебры и геометрии на уровне средней

школы, даже если вы ее давно окончили. Эта книга предназначена для чтения за клавиатурой. Наибольшую пользу от нее вы получите, если будете выполнять представленные в ней примеры и упражнения.

Структура издания

Глава 1 приглашает вас в мир математики. Она охватывает некоторые важные области применения этой науки в программировании, знакомит с темами, которые будут обсуждаться в книге, и объясняет, насколько полезным инструментом может быть программирование для изучающих математику. Далее книга делится на три части.

- Часть I посвящена векторам и линейной алгебре.
  - Глава 2 посвящена векторной математике в двухмерном пространстве с упором на использование координат в двухмерной графике. Здесь же рассматриваются некоторые основы тригонометрии.
  - Глава 3 дополняет предыдущую обсуждением трехмерных пространств, точки которых имеют три координаты. Здесь будут представлены понятия скалярного и векторного произведений, которые можно применять для измерения углов и отображения трехмерных моделей.
  - Глава 4 знакомит с линейными преобразованиями — функциями, принимающими и возвращающими векторы, которые создают определенные геометрические эффекты, такие как поворот или отражение.
  - Глава 5 знакомит с матрицами — массивами чисел, которые могут кодировать линейное векторное преобразование.
  - Глава 6 обобщает принципы операций с двух- и трехмерными векторами до *произвольного* числа измерений. Пространства таких векторов называют векторными пространствами. В качестве основного примера в ней используется обработка изображения с помощью векторной математики.
  - Глава 7 посвящена наиболее важной вычислительной задаче линейной алгебры — решению систем линейных уравнений. Здесь она задействуется в системе обнаружения столкновений в простой видеоигре.
- Часть II знакомит с численными методами и примерами их применения в физике.
  - Глава 8 вводит понятие скорости изменения функции. Она охватывает производные, определяющие скорость изменения функции, и интегралы, позволяющие восстановить функцию по скорости ее изменения.
  - В главе 9 рассматривается важный метод приближенного интегрирования, называемый методом Эйлера. Здесь игра, представленная в главе 7, будет дополнена движущимися и ускоряющимися объектами.

  - Глава 10 демонстрирует приемы манипулирования алгебраическими выражениями в коде, включая автоматический поиск формулы производной функции. Здесь вы познакомитесь с символьным программированием — подходом к выполнению математических операций в коде, отличным от подходов, используемых в других частях книги.
  - Глава 11 расширяет тему вычислительных методов до двух измерений, определяя операцию градиента и показывая пример определения силового поля с ее помощью.
  - Глава 12 демонстрирует применение производных для поиска максимальных или минимальных значений функций.
  - Глава 13 показывает возможность представления звуковых волн в виде функций и их разложения на суммы других, более простых функций, называемых рядами Фурье. Здесь рассказывается, как реализовать на Python воспроизведение музыкальных нот и аккордов.
- Часть III объединяет приемы из первых двух частей для представления некоторых важных принципов машинного обучения.
  - Глава 14 рассказывает, как методом линейной регрессии аппроксимировать двухмерные данные прямой линией. Примером в этой главе служит поиск функции, которая наиболее точно предсказывает цену подержанного автомобиля по его пробегу.
  - Глава 15 рассматривает другую задачу машинного обучения — определение модели автомобиля на основе некоторых данных о нем. Выяснение того, какой объект представлен точкой данных, называется классификацией.
  - Глава 16 показывает, как спроектировать и реализовать нейронную сеть — особый вид математической функции — и использовать ее для классификации изображений. В этой главе применяется то, что рассматривалось почти во всех предыдущих главах.

Каждая глава должна быть понятна, если вы прочитали и поняли предыдущие главы. Необходимость описания идей в определенном порядке обусловлена тем, что они могут иметь самые разные практические применения. Надеюсь, что разнообразие примеров сделает чтение книги увлекательным занятием и покажет вам широкий спектр практических применений математики.

О ПРИМЕРАХ КОДА

В этой книге темы представлены (я надеюсь) в логическом порядке. Концепции, описанные в главе 2, используются как основа в главе 3, на тех, что рассмотрены в главах 2 и 3, базируется глава 4 и т. д. Программный код не всегда пишется по порядку. То есть самые простые концепции в готовой компьютерной

программе не всегда находятся в первых строках первого файла исходного кода. Это различие затрудняет представление исходного кода книги в понятной форме.

Я попытался решить эту проблему за счет включения в каждую главу пошагового файла с кодом в виде блокнота Jupyter. Блокнот Jupyter можно считать аналогом записи интерактивного сеанса Python со встроенными визуальными элементами, такими как графики и изображения. Используя блокнот Jupyter, вы вводите какой-то код, выполняете его, а позже переопределяете в ходе сеанса по мере реализации своих идей. Блокнот для каждой главы включает код для каждого раздела и подраздела, который выполняется в том же порядке, что и в книге. Это означает, что вы можете опробовать примеры во время чтения. Вам не нужно добираться до конца главы, где код станет достаточно полным для выполнения. В приложении А показано, как настроить Python и Jupyter, а в приложении Б перечислены некоторые функции Python, полезные для начинающих изучать этот язык.

Эта книга содержит множество примеров исходного кода как в отдельных листингах, так и прямо внутри текста. В обоих случаях исходный код оформлен `моноширинным шрифтом`, чтобы его легко было отличить от обычного текста.

Кроме того, комментарии, имеющиеся в исходном коде, удалены из листингов, если они описываются в тексте. Многие листинги сопровождаются отдельными описаниями, подчеркивающими наиболее важные понятия. Если в исходном коде в репозитории книги будут обнаружены и исправлены какие-то ошибки, то там же, в репозитории, я добавлю примечания, объясняющие любые отличия от кода, напечатанного в книге.

В некоторых случаях примерами служат целые сценарии на Python, а не ячейки в блокноте Jupyter. Эти сценарии можно запускать в командной строке, например, командой `python script.py`, или в ячейке блокнота Jupyter командой `!python script.py`. Я включил ссылки на такие сценарии в некоторые блокноты Jupyter, чтобы в процессе чтения вы могли находить и опробовать соответствующие файлы с исходным кодом.

На протяжении всей книги я неизменно придерживаюсь одного соглашения: примеры выполнения отдельных команд Python начинаются с приглашения к вводу `>>>`, которое используется также в интерактивной оболочке Python. Я настоятельно рекомендую применять Jupyter вместо интерактивной оболочки Python, но в любом случае строки, начинающиеся с `>>>`, — это ввод, а остальные строки — вывод. Вот пример блока кода, представляющего вычисление выражения «2 + 2» в интерактивной оболочке Python:

```
>>> 2 + 2
4
```

В следующем примере отсутствует последовательность символов `>>>`, так что этот пример содержит обычный код на Python, а не последовательность ввода и вывода:

```
def square(x):
    return x * x
```

В книге приводятся сотни упражнений для закрепления пройденного материала, а также мини-проекты — либо более сложные, либо требующие более творческого подхода, либо знакомящие с новыми понятиями. Большинство упражнений и мини-проектов предлагают решить некоторые математические задачи с использованием кода на Python. Я дал решения почти всех, кроме некоторых мини-проектов, реализация которых не ограничивается временными рамками. Решения приводятся в блокноте Jupyter для соответствующей главы.

Код примеров из этой книги доступен для загрузки на веб-сайте издательства Manning (https://www.manning.com/books/math-for-programmers), а также в репозитории GitHub (https://github.com/orlandpm/math-for-programmers).

ОТ ИЗДАТЕЛЬСТВА

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Об авторе

Пол Орланд (Paul Orland) — предприниматель, программист и математик-энтузиаст. Поработав инженером-программистом в Microsoft, он основал компанию Tachyus, занимающуюся прогностической аналитикой в сфере оптимизации производства энергии в нефтегазовой отрасли. Как технический директор и основатель Tachyus, Пол руководил разработкой программного обеспечения для машинного обучения и моделирования физических процессов, а позже, как генеральный директор, расширил сферу влияния компании, найдя клиентов на пяти континентах. Пол имеет степень бакалавра по математике, полученную в Йельском университете, и степень магистра по физике, полученную в Вашингтонском университете. Его тотемное животное — лобстер.

Иллюстрация на обложке

На обложку книги помещена иллюстрация, подписанная *Femme Laponne* — женщина из Лапландии, ныне Сапми, которая включает части северной Норвегии, Швеции, Финляндии и России. Иллюстрация взята из каталога костюмов жителей разных стран, изданного Жаком Грассе де Сен-Совером (Jacques Grasset de Saint-Sauveur; 1757–1810) в 1797 году во Франции под названием *Costumes de Différents Pays*. Все иллюстрации в каталоге тщательно прорисованы и раскрашены вручную. Богатое разнообразие коллекции Грассе де Сен-Совера напоминает нам о том, насколько отличными друг от друга были культурные традиции городов и регионов мира всего 200 лет назад. Изолированные друг от друга люди говорили на разных диалектах и языках. Встретив человека на улице, по его одежде легко было определить, где он живет, чем зарабатывает на жизнь или какое положение в обществе занимает.

С тех пор стиль одежды сильно изменился, исчезло разнообразие, характеризующее различные области и страны. В настоящее время трудно различить по одежде даже жителей разных континентов, не говоря уже о жителях разных городов, регионов или стран. Мы заменили культурное многообразие более разнообразной личной жизнью и, безусловно, более разнообразной и интересной интеллектуальной жизнью.

Мы в издательстве Manning славим изобретательность и предприимчивость компьютерного бизнеса обложками книг, изображающими богатство региональных различий двухвековой давности, оживших благодаря иллюстрациям Грассе де Сен-Совера.

1 Математика в программном коде

В этой главе

- ✓ Решение финансовых задач с помощью математики и программного обеспечения.
- ✓ Как избежать распространенных ошибок при изучении математики.
- ✓ От программирования к пониманию математики, основываясь на интуиции.
- ✓ Python как мощный и расширяемый калькулятор.

Математика подобна бейсболу, поэзии или хорошему вину. Одни настолько увлечены этой наукой, что посвящают ей всю свою жизнь, другим же кажется, что они просто не понимают ее. Вероятно, после 11 лет изучения математики в школе вы уже причислили себя к той или иной группе людей.

Представьте, что в школе вы бы изучали виноделие подобно тому, как изучали математику. Мне бы не понравилось, если бы мне читали лекции о сортах винограда и методах ферментации по часу в день шесть дней в неделю. Будь такой предмет в школе, то, может быть, мне приходилось бы выпивать по три-четыре стакана, чтобы выполнить домашнее задание. С одной стороны, это мог бы быть восхитительный образовательный опыт, но с другой — идти следующим утром на занятия с тяжелой головой — сомнительное удовольствие. Опыт, который я получил на уроках математики, был примерно таким же, и это на какое-то время отталкивало меня от предмета.

Проще всего решить, что вы либо созданы для математики, либо нет. Если вы уже верите в себя и хотите начать учиться — здорово! В остальном эта глава предназначена для тех, кто настроен менее оптимистично. Страх перед математикой настолько распространен, что даже назван *математической тревожностью (math anxiety)*. Я надеюсь рассеять ваше беспокойство и показать, что математика может быть очень интересным увлечением. Все, что вам нужно, — это правильные инструменты и правильный настрой.

Основным инструментом обучения в этой книге станет язык программирования Python. Полагаю, что когда вы изучали математику в школе, то учились по формулам, написанным на доске, а не по компьютерному коду. Это беда, потому что язык программирования высокого уровня намного мощнее школьной доски и намного универсальнее любого дорогого калькулятора, который вы могли бы использовать. Преимущество изучения математики на примерах в программном коде состоит в том, что идеи в этом случае выражаются достаточно точно, чтобы их мог понять компьютер, и нет необходимости спорить по поводу значений новых символов.

Как и при изучении любого нового предмета, лучший способ настроиться на успех — *захотеть* учиться. Тому может быть множество причин. Вас может заинтриговать красота математических понятий, или вам может доставлять удовольствие решение головоломных математических задач. Возможно, вы мечтаете создать приложение или игру, а для этого нужно реализовать математические вычисления в коде. Но я оставлю все эти причины в стороне и сосредоточусь на более прагматичной мотивации — решение математических задач с помощью программного обеспечения может принести вам много денег.

1.1. РЕШЕНИЕ ФИНАНСОВЫХ ЗАДАЧ С ПОМОЩЬЮ МАТЕМАТИКИ И ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

На уроках математики от нерадивых учеников часто можно услышать вопрос: «Где в реальной жизни мне могут пригодиться эти знания?» Когда я учился в школе, учителя говорили нам, что математика поможет добиться профессионального успеха и заработать деньги. Я думаю, что они были правы, хотя иногда и приводили неправильные примеры. Например, я не рассчитываю банковские проценты вручную, как и мой банк. Может быть, если бы я стал геодезистом на стройплощадке, как предположил мой учитель тригонометрии, то использовал бы синусы и косинусы каждый день, чтобы заслужить свою зарплату.

Как оказалось, примеры из школьных учебников не так уж и практичны. И все же в реальной жизни есть примеры ошеломляюще прибыльного применения математики. Многие из них заключаются в воплощении правильной математической идеи в пригодном для использования программном обеспечении. Некоторыми из таких примеров я хочу с вами поделиться.

1.1.1. Прогнозирование движения финансового рынка

Все мы слышали легенды о биржевых трейдерах, которые зарабатывают миллионы долларов и умудряются покупать и продавать нужные акции в нужное время. По фильмам, которые я видел, у меня сложился стереотипный образ трейдера как мужчины средних лет в костюме, который кричит на своего брокера по мобильному телефону и разъезжает на спортивной машине. Возможно, когда-то это и соответствовало действительности, но сейчас все изменилось.

В офисах, разбросанных по всему Манхэттену, скрываются тысячи людей, которых называют *биржевыми аналитиками*. Они разрабатывают математические алгоритмы для автоматизации торговли акциями и получения прибыли. Они не носят костюмы и не кричат в мобильные телефоны, но я уверен, что у многих из них очень хорошие спортивные автомобили.

Но как биржевой аналитик может написать программу, которая автоматически зарабатывает деньги? Ответы на этот вопрос охраняются как самые ценные коммерческие секреты, но можете быть уверены: в них много математики. Рассмотрим короткий пример, чтобы понять, как может работать автоматизированная стратегия торговли.

Акции — это разновидность финансовых активов, представляющих доли участия в компаниях. Когда рынок понимает, что дела у компании идут хорошо, ее акции растут в цене — их покупка становится более дорогостоящей, а продажа — более прибыльной. Цены на акции меняются постоянно и хаотично. На рис. 1.1 показано, как может выглядеть график изменения цены акции в течение торгового дня.



Рис. 1.1. Типичный график изменения цены акции с течением времени

Если купить 1000 этих акций по 24 доллара примерно на 100-й минуте и продать по 38 долларов на 400-й минуте, то можно заработать 14 000 долларов всего за один день. Неплохо! Однако проблема в том, что для этого нужно заранее знать, что акции вырастут и что 100-я и 400-я минуты являются лучшими моментами для покупки и продажи соответственно. Конечно, никто не сможет точно предсказать моменты, когда цена будет минимальной или максимальной, но можно попробовать найти относительно хорошее время для покупки и продажи в течение дня. Посмотрим, как эту задачу решить математически.

Мы могли бы определить, движется ли цена акций вверх или вниз, найдя линию наилучшего соответствия, которая примерно соответствует направлению движения цены. Этот процесс называется *линейной регрессией*, и мы рассмотрим его в части III. Основываясь на изменчивости данных, можно рассчитать еще две линии выше и ниже линии наилучшего соответствия, которые ограничивают область колебания цены. Как показано на рис. 1.2, эти три линии, наложенные на график колебания цены, довольно хорошо отражают общий тренд.

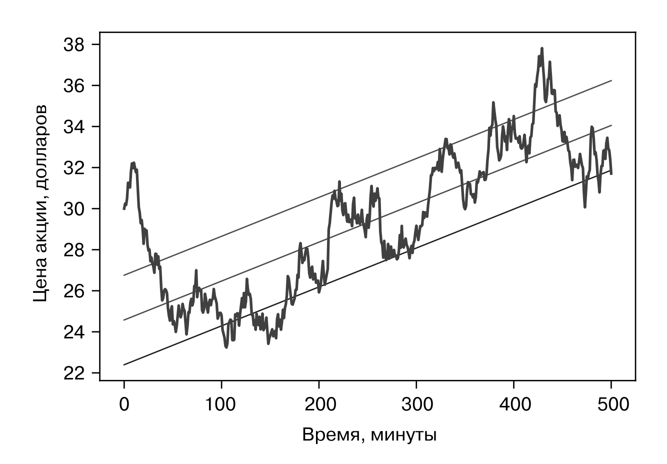


Рис. 1.2. Использование линейной регрессии для выявления тренда изменения цены на акции

Получив математическое понимание движения цены, можно написать код для автоматической покупки, когда цена колеблется возле нижней точки тренда, и продажи, когда она возвращается вверх. В частности, программа может подключаться к фондовой бирже по сети и покупать 100 акций, когда цена пересекает нижнюю линию, и продавать 100 акций, когда цена пересекает верхнюю линию. Рисунок 1.3 иллюстрирует одну из таких прибыльных сделок: покупка по цене около 27,8 доллара и продажа по цене около 32,6 доллара дают заработок 480 долларов в час.

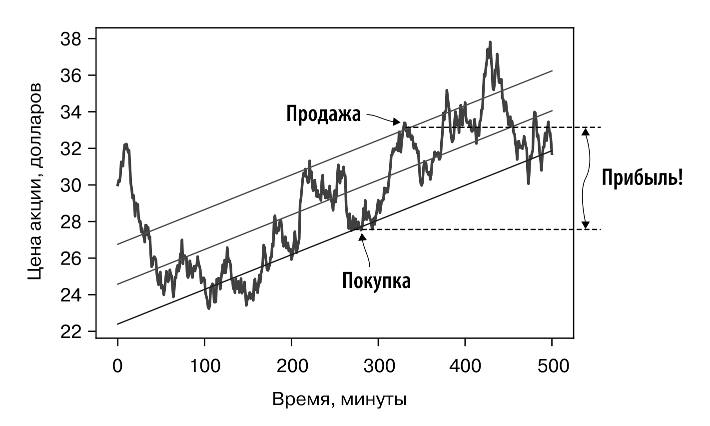


Рис. 1.3. Покупка и продажа в соответствии с установленными нами правилами обеспечивают прибыль

Я не берусь утверждать, что продемонстрировал полноценную или хотя бы жизнеспособную стратегию, но дело в том, что, имея правильную математическую модель, можно автоматически получать прибыль. Сейчас множество программ строят и обновляют модели, оценивающие прогнозируемую динамику изменения стоимости акций и других финансовых инструментов. Если вы напишете такую программу, то сможете наслаждаться отдыхом, в то время как она будет приносить вам деньги!

1.1.2. Поиск выгодной сделки

Наверняка у вас не настолько туго набитые карманы, чтобы пускаться в рискованные операции с акциями. Но математика все равно может помочь заработать и сэкономить деньги в других сферах, таких, например, как покупка подержанного автомобиля. С новой машиной все просто: если два дилера продают одинаковые автомобили, то вы, очевидно, пойдете к тому, который предложит наименьшую цену. Однако при покупке подержанного автомобиля приходится оценивать больше факторов — не только цену, но также пробег и год выпуска. Вы можете учитывать даже продолжительность нахождения конкретного подержанного автомобиля на рынке как косвенную оценку его качества: чем дольше он продается, тем более подозрительным кажется.

В математике объекты, которые можно описать с помощью упорядоченных списков чисел, называются *векторами*, и существует целая область математики, называемая линейной алгеброй, посвященная их изучению. Например, подержанный автомобиль может характеризоваться *четырехмерным* вектором, то есть четверкой чисел (2015, 41 429, 22,27, 16 980). Этот вектор содержит год выпуска модели, пробег, количество дней на рынке и запрашиваемую цену соответственно. У моего друга есть сайт CarGraph.com, на котором собраны данные о выставленных на продажу подержанных автомобилях. На момент написания

этих строк в списке насчитывался 101 автомобиль Toyota Prius, выставленный на продажу, и для каждого были предоставлены некоторые или все четыре элемента данных. Кроме того, оправдывая свое название, сайт предлагает визуальное представление данных в виде графика (рис. 1.4). Визуализировать четырехмерные объекты трудно, но если выбрать два измерения, такие как цена и пробег, то их можно изобразить в виде точек на точечной диаграмме.

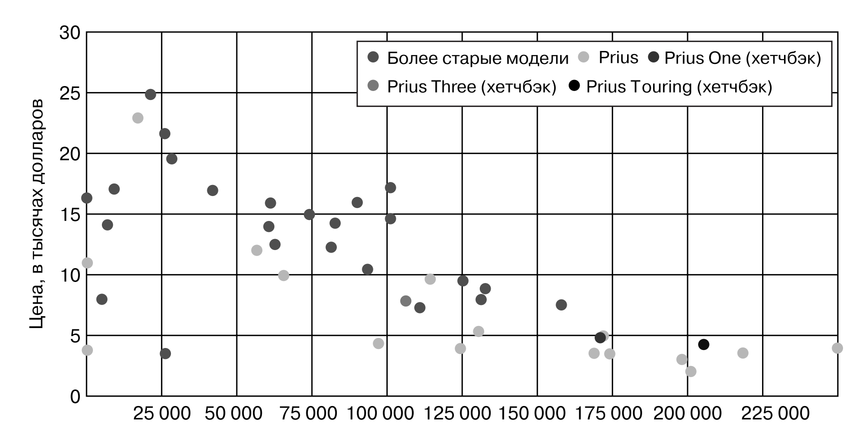


Рис. 1.4. Соотношение «цена/пробег» для автомобилей Toyota Prius, выставленных на продажу на сайте CarGraph.com

Было бы интересно попробовать построить линию тренда. Каждая точка на этом графике представляет чье-то мнение о справедливой цене, поэтому линия тренда объединит эти мнения в более надежную цену при любом пробеге. На рис. 1.5 я использовал *экспоненциальную* кривую вместо прямой и исключил из расчетов некоторые почти новые автомобили, продающиеся по цене ниже розничной.

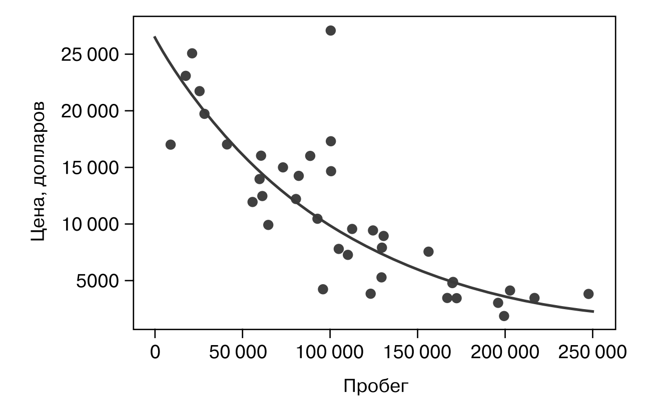


Рис. 1.5. Аппроксимация экспоненциальной кривой соотношения «цена/пробег» для подержанных автомобилей Toyota Prius

Для большего удобства я преобразовал значения пробега в десятки тысяч миль, поэтому пробег 5 соответствует 50 000 миль. Обозначив цену p, а пробег m, я вывел уравнение кривой наилучшего приближения:

$$p = 26\,500 \cdot 0{,}905^{m}. \qquad (1.1)$$

Как показывает уравнение (1.1), в среднем цена подержанного автомобиля равна 26 500 долларов, умноженных на 0,905 в степени величины пробега. Подставив значения в уравнение, я выяснил, что, располагая 10 000 долларов, могу купить Prius с пробегом около 97 000 миль (рис. 1.6). Если считать эту кривую отражением *справедливой* цены, то автомобили ниже этой линии можно рассматривать как выгодные предложения.

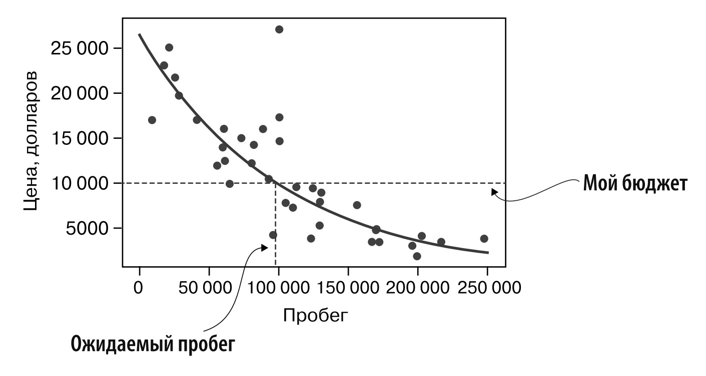


Рис. 1.6. Определение пробега автомобиля Prius, который можно приобрести при бюджете 10 000 долларов

Но уравнение (1.1) не только позволяет определить выгодность сделки. Оно рассказывает о том, как обесцениваются автомобили. Первое число в уравнении — 26 500 долларов — соответствует цене автомобиля с нулевым пробегом. Она удивительно близка к розничной цене нового Prius. Если бы наилучшей аппроксимацией тренда была прямая линия, то это означало бы, что стоимость Prius уменьшается на фиксированную сумму с каждой пройденной милей. Однако экспоненциальная форма кривой говорит о том, что стоимость уменьшается на фиксированный процент. Согласно этому уравнению, проехав 10 000 миль, Prius будет стоить 0,905, или 90,5 %, от первоначальной цены. После 50 000 миль пробега его цену нужно умножить на коэффициент $0{,}905^5 = 0{,}607$, то есть стоимость автомобиля составит 61 % от первоначальной цены.

Чтобы построить график, изображенный на рис. 1.6, я реализовал функцию `price(mileage)` на Python, которая принимает величину пробега (в десятках тысяч миль) и возвращает наиболее вероятную цену. Результаты вычислений `price(0) - price(5)` и `price(5) - price(10)` говорят мне, что первые и вторые 50 000 миль снижают стоимость примерно на 10 000 и 6300 долларов соответственно.

Аппроксимация тренда прямой линией означала бы, что стоимость автомобиля уменьшается на фиксированную величину 0,1 доллара за милю. То есть через каждые 50 000 миль пробега стоимость должна уменьшаться на одни и те же 5000 долларов. Но здравый смысл подсказывает, что первые мили пробега новой машины — самые дорогие, и экспоненциальная функция (уравнение (1.1)) хорошо согласуется с этим, а линейная модель — нет.

Но давайте не будем забывать, что это лишь *двухмерный* анализ. Мы построили математическую модель, использующую две характеристики из четырех, описывающих каждый автомобиль. В части I вы познакомитесь с векторами различных размерностей и узнаете, как манипулировать многомерными данными. В части II мы рассмотрим различные виды функций, такие как линейные и экспоненциальные, и сравним их, проанализировав скорости изменения. Наконец, в части III вы узнаете, как строить математические модели, которые включают *все* измерения, имеющиеся в наборе данных, и позволяют получить более точную картину.

1.1.3. Трехмерная графика и анимация

Многие из самых известных и финансово успешных программных проектов так или иначе обрабатывают многомерные данные, часто *трехмерные*. Здесь я имею в виду трехмерные анимационные фильмы и видеоигры, кассовые сборы которых исчисляются миллиардами долларов. Например, программное обеспечение Pixar для трехмерной анимации помогло заработать более 13 млрд долларов. Серия трехмерных экшен-игр *Call of Duty*, выпущенных компанией Activision, принесла ей более 16 млрд долларов, а *Grand Theft Auto V*, выпущенная компанией Rockstar, — 6 млрд долларов.

Каждый из этих проектов основан на вычислениях с трехмерными векторами, или тройками чисел, вида $v = (x, y, z)$. Тройки чисел достаточно, чтобы найти точку в трехмерном пространстве относительно контрольной точки, называемой началом координат. Как показано на рис. 1.7, каждое из трех чисел говорит, как далеко нужно пройти в одном из трех перпендикулярных направлений, чтобы достичь заданной точки.

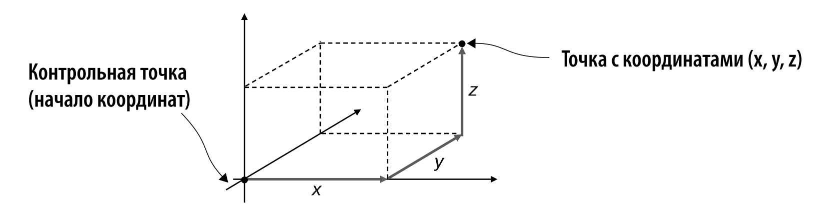


Рис. 1.7. Определение точки в трехмерном пространстве с помощью вектора из трех чисел: *x*, *y* и *z*

Любой трехмерный объект, от рыбы-клоуна в игре «В поисках Немо» до авианосца в *Call of Duty*, можно определить в компьютере как набор трехмерных векторов. В программе каждый из этих объектов выглядит как список троек чисел типа `float`. Три тройки чисел определяют три точки в пространстве, которые могут представлять треугольник (рис. 1.8), например:

```
triangle = [(2.3,1.1,0.9), (4.5,3.3,2.0), (1.0,3.5,3.9)]
```

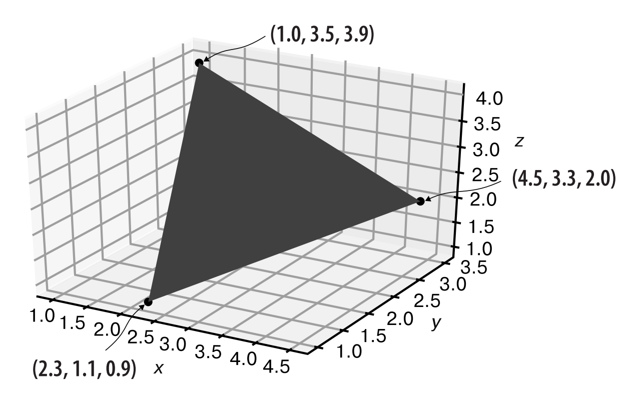


Рис. 1.8. Трехмерный треугольник, образованный тремя тройками чисел, обозначающими координаты вершин

Комбинируя множество треугольников, можно определить поверхность трехмерного объекта, и чем больше треугольников меньшего размера использовать, тем более гладким получится объект. На рис. 1.9 показаны варианты определения трехмерной сферы с помощью все большего числа треугольников все меньшего и меньшего размера.

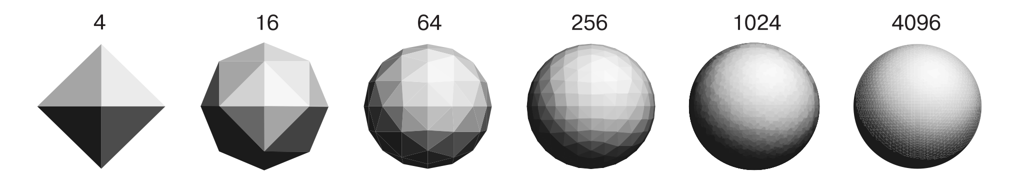


Рис. 1.9. Трехмерные сферы, образованные разным количеством треугольников

В главах 3 и 4 вы узнаете, как использовать трехмерную векторную математику для преобразования трехмерных моделей в двухмерные изображения с тенями, как на рис. 1.9. Кроме того, трехмерные модели должны быть гладкими, чтобы иметь реалистичный вид в игре или фильме, и должны двигаться и изменяться реалистичным образом. Это означает, что рисованные объекты должны подчиняться законам физики, которые тоже выражаются в терминах трехмерных векторов.

Предположим, что вы программист, занимающийся разработкой *Grand Theft Auto V*, и хотите включить в игру поддержку, например, стрельбы из базуки по вертолету. Снаряд, вылетающий из базуки, начинает полет в точке местоположения главного героя, затем его позиция меняется со временем. Для обозначения различных позиций, которые занимает снаряд на протяжении полета, можно использовать числовые индексы, начиная с $v_0 = (x_0, y_0, z_0)$. С течением времени снаряд достигает новых позиций, обозначаемых векторами $\mathbf{v}_1 = (x_1, y_1, z_1)$, $\mathbf{v}_2 = (x_2, y_2, z_2)$ и т. д. Скорость изменения значений x, y и z определяется направлением выстрела и скоростью движения снаряда. Кроме того, скорость может меняться со временем — снаряд увеличивает свою позицию по оси z с уменьшающейся скоростью из-за непрерывного действия силы тяжести, направленной вертикально вниз (рис. 1.10).

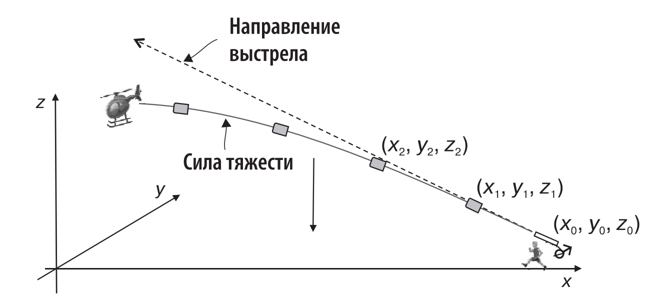


Рис. 1.10. Вектор, описывающий положение снаряда, изменяется с течением времени и определяется начальной скоростью снаряда и силой тяжести

Любой опытный геймер скажет вам, что нужно целиться немного выше вертолета, чтобы попасть в него! Для моделирования физики необходимо знать, какие силы воздействуют на объекты и какие изменения с течением времени они вызывают. Математика непрерывных изменений называется *математическим анализом*, а законы физики обычно выражаются в терминах объектов матанализа, называемых *дифференциальными уравнениями*. Подробнее об анимации трехмерных объектов вы узнаете в главах 4 и 5, а о моделировании физических законов с использованием матанализа — в части II.

1.1.4. Моделирование физического мира

Заявление о том, что математическое программное обеспечение создает реальную финансовую ценность, — не просто предположение, доказательством может служить моя карьера. В 2013 году я основал компанию Tachyus, которая занимается разработкой программного обеспечения для оптимизации добычи нефти и газа. Наше программное обеспечение использует математические модели потоков нефти и газа под землей, помогающие добывать их более

эффективно и прибыльно. С помощью полученной информации наши клиенты смогли сэкономить миллионы долларов за счет снижения затрат и увеличения добычи.

Чтобы объяснить, как работает наше программное обеспечение, необходимо знать хотя бы основные термины нефтедобычи. В земле бурят отверстия, называемые *скважинами*, пока не будет достигнут слой пористой породы, содержащей нефть. Этот слой породы, богатый нефтью, называется *резервуаром*. Нефть выкачивается на поверхность, а затем продается нефтеперерабатывающим предприятиям, превращающим ее в нефтепродукты, которые мы используем каждый день. На рис. 1.11 показана упрощенная схема нефтяного месторождения.

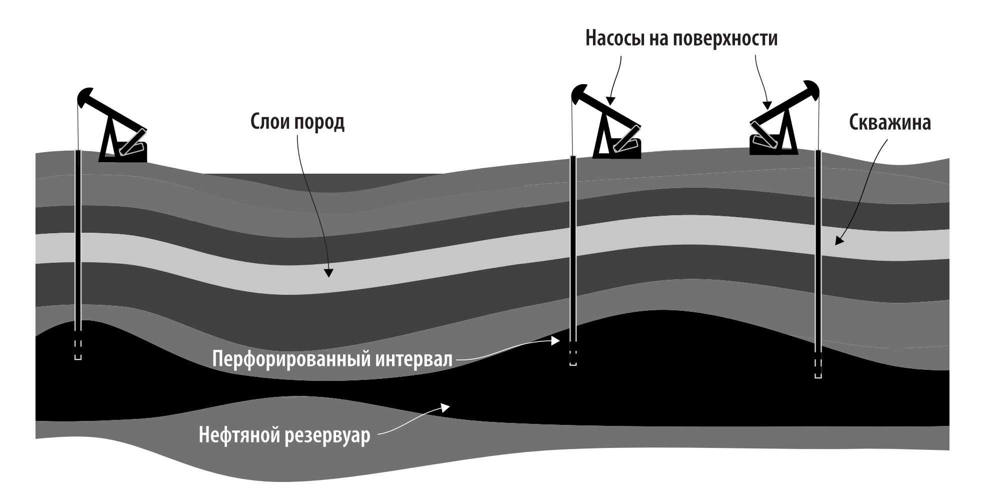


Рис. 1.11. Упрощенная схема нефтяного месторождения

В последние несколько лет наблюдались значительные колебания цен на нефть, но для цели дальнейшего обсуждения допустим, что нефть стоит 50 долларов за баррель, где баррель — это единица объема, равная 42 галлонам, или примерно 159 литрам. Если предположить, что в результате бурения скважин и эффективной откачки компания способна добывать 1000 баррелей нефти в день (объем нескольких плавательных бассейнов на заднем дворе), то ее годовой доход будет исчисляться десятками миллионов долларов и увеличение эффективности даже на несколько процентов может приносить значительную дополнительную прибыль.

Основной вопрос, волнующий нефтедобытчиков: что происходит под землей, как движется нефть? Это сложный вопрос, но на него можно ответить более или менее точно, решая дифференциальные уравнения. Роль переменной здесь играет не положение снаряда, а расположение, давление и дебит жидкостей в пласте. Дебит жидкостей — это функция особого вида, которая возвращает

вектор, называемый *векторным полем*. Текучая среда может распространяться с любой скоростью в любом из направлений в трехмерном пространстве, и эти направление и скорость могут различаться в разных местах внутри резервуара.

Подбирая наиболее вероятные значения для некоторых из этих параметров, мы можем прогнозировать скорость потока жидкостей через пористую горную породу, например песчаник, используя дифференциальное уравнение, называемое *законом Дарси*. На рис. 1.12 показана формула, выражающая закон Дарси, — не волнуйтесь, если какие-то символы окажутся вам незнакомыми! Функция **q**, представляющая скорость потока, выделена жирным, чтобы показать, что она возвращает векторное значение.

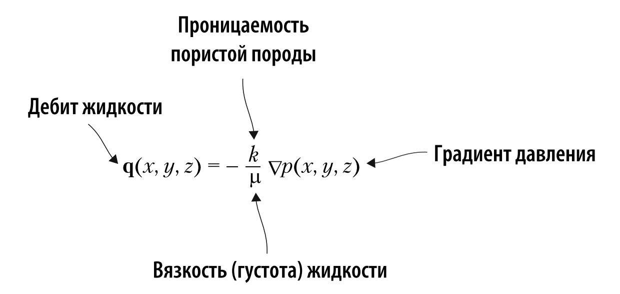


Рис. 1.12. Формула, выражающая закон Дарси, который описывает течение жидкостей через пористые горные породы

Наиболее важная часть этого уравнения — символ в виде треугольника, обращенного вершиной вниз, который представляет *оператор градиента* в векторном анализе. Градиент функции давления $p(x, y, z)$ в данной пространственной точке (x, y, z) — это трехмерный вектор $\mathbf{q}(x, y, z)$, указывающий направление и скорость увеличения давления в этой точке. Знак «минус» обозначает, что трехмерный вектор скорости потока имеет *противоположное* направление. Это уравнение говорит языком математики, что жидкость течет из областей с высоким давлением в области с низким давлением.

Отрицательные градиенты часто встречаются в физике. Их можно рассматривать как свидетельство того, что природа всегда стремится перейти в состояние с более низкой потенциальной энергией. Потенциальная энергия мяча на холме зависит от высоты холма h в любой его точке x. Если представить, что высота холма задана функцией $h(x)$, то ее градиент направлен вверх, а мяч будет стремиться скатиться вниз — в прямо противоположном направлении (рис. 1.13).

В главе 11 вы узнаете, как вычислять градиенты. Там я покажу, как применять их для моделирования законов физики, а также для решения других математических задач. Кроме того, градиент — одно из самых важных математических понятий в машинном обучении.



Рис. 1.13. Положительный градиент направлен вверх, а отрицательный — вниз

Я надеюсь, что эти примеры показались вам более убедительными и реалистичными, чем примеры, приводившиеся на уроках математики в школе. Возможно, сейчас вы убедились, что эти математические понятия достойны изучения, но беспокоитесь, что они могут оказаться слишком сложными для вас. Изучение математики — трудная задача, особенно в одиночку. Поэтому, чтобы упростить ее, поговорим о некоторых возможных ловушках, и о том, как я помогу вам избежать их в этой книге.

1.2. КАК НЕ НАДО УЧИТЬ МАТЕМАТИКУ

На рынке есть много книг по математике, но не все они одинаково полезны. У меня есть немало друзей-программистов, которые пытались изучать математические концепции, подобные описанным в предыдущем разделе, просто из любопытства или из-за стремления продвинуться по карьерной лестнице. Так вот, используя традиционные учебники по математике в качестве основного ресурса, они часто оказывались в тупике и сдавались. Вот как выглядит типичная история неудачного изучения математики.

1.2.1. Джейн решила подучить математику

Моя (вымышленная) подруга Джейн, веб-разработчик полного цикла, трудится в небольшой технологической компании в Сан-Франциско. В колледже Джейн изучала информатику и математику по самой обычной программе и начинала карьеру как менеджер по продукту. За последние 10 лет она изучила программирование на Python и JavaScript и смогла перейти в отдел разработки программного обеспечения. На новой работе она считается одним из самых

способных программистов, так как умеет создавать базы данных, веб-сервисы и пользовательские интерфейсы для обслуживания клиентов. Как видите, она большая умница!

Постепенно Джейн поняла, что изучение науки о данных поможет ей разрабатывать и внедрять новые возможности и улучшить качество обслуживания клиентов. Большую часть времени по пути на работу Джейн читает блоги и статьи о новых технологиях, а недавно ее поразили несколько статей на тему глубокого обучения. В одной из них рассказывалось о модели глубокого обучения AlphaGo, созданной в Google, сумевшей обыграть в настольную игру го лучших игроков в мире. В другой статье были показаны потрясающие картины в технике импрессионистов, созданные из обычных изображений, опять же с использованием модели глубокого обучения.

Прочитав эти статьи, Джейн услышала, что Маркус, друг ее друга, получил работу по исследованию технологий глубокого обучения в крупной технологической компании. Маркус якобы получает более 400 000 долларов в год в виде зарплаты и акций. Тогда Джейн задумалась о следующей ступени в своей карьере: разве может быть что-то лучше, чем увлекательная и прибыльная работа?

Немного покопавшись в Интернете, Джейн нашла авторитетный (и бесплатный!) ресурс — книгу «Глубокое обучение» Яна Гудфеллоу и др.[1]. Введение было очень похоже на технические статьи в блоге, к которым она привыкла, и это еще больше привлекло ее к изучению темы. Но постепенно читать книгу становилось все труднее. В первой главе авторы представили все необходимые математические понятия и ввели множество терминов и обозначений, которых Джейн раньше никогда не видела. Она пробежала глазами эту главу и попыталась перейти к сути, но ей становилось все сложнее.

Джейн решила, что нужно отложить изучение искусственного интеллекта (ИИ) и глубокого обучения до тех пор, пока она не поднатореет в математике. К счастью, в главе, посвященной математике, среди прочего был рекомендован учебник по линейной алгебре для студентов, которые никогда раньше не изучали этот предмет. Она разыскала этот учебник Георгия Шилова (Georgi Shilov) *Linear Algebra* (Dover, 1977) и обнаружила, что все его 400 страниц так же густо усеяны незнакомыми ей понятиями, как и книга «Глубокое обучение».

Проведя день за чтением заумных теорем о таких понятиях, как числовые поля, определители и алгебраические дополнения, она решила отказаться от своей затеи. Она не могла понять, как эти концепции помогут ей написать программу, способную выиграть у человека в настольную игру или нарисовать картину, и ей больше не хотелось тратить десятки часов на изучение этого сухого материала.

---

1 *Курвилль А., Гудфеллоу Я., Бенджио И.* Глубокое обучение.

Спустя какое-то время мы с Джейн встретились, чтобы поболтать за чашечкой кофе. Она рассказала мне о трудностях, с которыми столкнулась при чтении литературы по ИИ из-за того, что не знала линейной алгебры. В последнее время я часто слышу похожие жалобы: «Я пытаюсь читать о [новой технологии], но, похоже, сначала мне нужно изучить [тему по математике]».

Она предприняла замечательную попытку: нашла лучший ресурс по предмету, который хотела изучить, и искала ресурсы для получения необходимых начальных знаний, которых ей не хватало. Но доведя этот подход до логического завершения, она оказалась в изматывающем «прочесывании» технической литературы.

1.2.2. Кропотливое изучение учебников по математике

Учебники по математике для вузов, такие как учебник по линейной алгебре, который подобрала Джейн, как правило, однотипны. Все разделы соответствуют одному формату: вводят новую терминологию, обозначают некоторые факты, называемые *теоремами*, с использованием этой терминологии, а затем доказывают верность этих теорем.

С виду как будто все логично: вы вводите понятие, о котором пойдет речь дальше, формулируете какие-то выводы, а затем обосновываете их. Но почему тогда так трудно читать учебники по математике?

Проблема в том, что сама математика развивается совсем не так. Когда исследователь придумывает новую математическую идею, ему может потребоваться время на эксперименты, прежде чем он найдет правильные определения. Я думаю, что большинство профессиональных математиков описали бы свои шаги так.

1. Придумай *игру*. Например, начни играть с некоторыми математическими объектами, пытаясь перечислить их все, найти общие закономерности или объект с определенным свойством.
2. Сформулируй некоторые *гипотезы*. Подумай об общих фактах, которые можно сообщить об игре, и убеди хотя бы себя в том, что они верны.
3. Разработай *точный язык* для описания игры и предположений. В конце концов, предположения ничего не значат, пока их нельзя высказать.
4. Наконец, добавив немного решимости и удачи, найди доказательство своей гипотезы, показывающее, почему она верна.

Главный урок, который можно вынести из этого процесса, заключается в том, что развитие глобальных идей начинается с их осмысления, а формализация откладывается на потом. Как только вы получите общее представление о развиваемой

идее, словарный запас и обозначения станут для вас преимуществом, а не отвлекающим фактором. Учебники по математике обычно написаны иначе, поэтому я рекомендую использовать их как справочники, а не для знакомства с новыми предметами.

Лучший способ изучения математики — не чтение традиционных учебников, а исследование идей и формулировка собственных выводов. Однако вам не хватит часов в сутках, чтобы самому все заново изобрести. Так как лучше поступить? Я выскажу вам свое скромное мнение, которым руководствовался, работая над этой нестандартной книгой по математике.

1.3. ИСПОЛЬЗОВАНИЕ НАТРЕНИРОВАННОГО ЛЕВОГО ПОЛУШАРИЯ

Эта книга адресована опытным программистам или желающим изучать программирование в процессе работы. Писать о математике для программистов — здорово, потому что тот, кто умеет писать код, уже натренировал левое полушарие, отвечающее за аналитическое мышление. Как мне кажется, лучший способ изучать математику — использовать язык программирования высокого уровня, и я полагаю, что в недалеком будущем это станет нормой в математических классах.

Есть несколько причин считать программистов хорошо подготовленными к изучению математики. Я перечисляю их здесь, не только чтобы польстить вам, но и чтобы напомнить, на какие из имеющихся у вас навыков можно опереться в своих математических исследованиях.

1.3.1. Использование формального языка

Один из первых сложных уроков, которые вы усвоили в программировании: код пишется совсем не так, как предложения на обычном языке. Если вы немного ошибетесь в грамматике, когда пишете записку другу, то вас почти наверняка поймут правильно. Но любая синтаксическая ошибка или неправильно написанный идентификатор в коде влекут за собой сбой в работе программы. В некоторых языках даже отсутствие точки с запятой в конце оператора, который в остальном верен, не позволяет запустить программу. В качестве еще одного примера рассмотрим два утверждения:

```
x = 5
5 = x
```

Любое из них можно прочитать как означающее, что символ x имеет значение 5. Но на языке Python эти два утверждения имеют *неодинаковые* значения

и синтаксически верным считается только первое. Оператор `x = 5` на языке Python — это инструкция присваивания значения 5 переменной x. А оператор `5 = x` интерпретируется как попытка присвоить значение x числу 5, что в принципе невозможно. Такая интерпретация может показаться чересчур формализованной, но это совершенно необходимо, чтобы написать правильную программу.

Еще один пример, сбивающий с толку начинающих программистов (да и опытных тоже!), — равенство ссылок. Если определить новый класс на языке Python и создать два идентичных экземпляра, они не будут равны:

```
>>> class A(): pass
...
>>> A() == A()
False
```

Казалось бы, два одинаковых выражения должны быть равны, но это правило не действует в языке Python, так как в сравнении участвуют разные экземпляры класса `A`, которые не считаются равными.

Будьте внимательны, работая с новыми математическими объектами, которые выглядят как известные вам, но ведут себя иначе. Например, если буквы A и B обозначают числа, то $AB = BA$. Но как вы узнаете в главе 5, это правило не всегда верно, если A и B не являются числами. Если буквами A и B обозначить матрицы, то произведения AB и BA будут иметь разные результаты. Может случиться так, что действительным может быть только одно из произведений или даже ни одного.

Чтобы написать код, недостаточно написать операторы с правильным синтаксисом. Положения, которые представляют операторы, должны иметь смысл. Если вы проявите такую же осторожность при написании математических утверждений, то быстрее будете находить свои ошибки. Еще лучше, если станете записывать свои математические утверждения в коде, тогда часть работы по их проверке возьмет на себя компьютер.

1.3.2. Создайте свой калькулятор

Калькуляторы активно используют на уроках математики, потому что они помогают проверить результаты, вычисленные вручную. Вы должны знать, сколько будет $6 \cdot 7$, без калькулятора, но нелишним будет убедиться, что ответ 42 — правильный, сверившись с калькулятором. Калькулятор поможет сэкономить время и позднее, когда математические понятия уже будут освоены. Если вы занимаетесь тригонометрией и вам нужно найти результат 3,14159 / 6, то калькулятор поможет быстро получить его, а вы можете лишние секунды поразмыслить над тем, что означает ответ. Чем шире возможности калькулятора, тем, теоретически, он должен быть полезнее.

Но иногда калькуляторы оказываются слишком сложными. Когда я перешел в старшие классы, мне понадобился графический калькулятор, и я получил TI-84. У него было около 40 кнопок, каждая из которых имела 2–3 режима. Я знал, как использовать от силы 20 из них, так что для меня это был слишком громоздкий инструмент. То же самое было, когда в первом классе я получил свой первый калькулятор. На нем было всего 15 кнопок, но я не знал, что делают некоторые из них. Если бы мне довелось создавать первый калькулятор для школьников, я бы сделал его похожим на изображенный на рис. 1.14.

Этот калькулятор имеет всего две кнопки. Одна из них вводит начальное значение 1, а другая выполняет переход к следующему числу. Такой калькулятор был бы подходящим инструментом для детей, которые учатся считать. (Мой пример может показаться смешным, но такие калькуляторы действительно можно купить! Обычно они механические и продаются под видом счетчиков-шагомеров.)

Освоив счет, вы захотите попрактиковаться в написании чисел и их сложении. Идеальный калькулятор на этом этапе обучения может иметь несколько дополнительных кнопок (рис. 1.15).



Рис. 1.14. Калькулятор для школьников, которые учатся считать



Рис. 1.15. Калькулятор, позволяющий вводить целые числа и складывать их

На этом этапе вам не нужны такие кнопки, как –, × или ÷. Даже решая задачи на вычитание, такие как 5 – 2, вы все равно сможете проверить свой ответ 3, вычислив с помощью калькулятора сумму 3 + 2 = 5. Точно так же сможете решать задачи на умножение, многократно складывая числа. Закончив изучение арифметических операций, можете перейти к использованию калькулятора, выполняющего все арифметические операции.

Я думаю, что идеальный калькулятор должен быть расширяемым, то есть давать возможность добавлять в него дополнительные функции по мере необходимости. Например, изучая новую математическую операцию, вы можете добавить на свой калькулятор соответствующую кнопку. Добравшись до алгебры, сможете заставить калькулятор понимать такие символы, как x или y, в дополнение к числам. Освоив матанализ, сможете научить калькулятор понимать другие математические функции.

Расширяемые калькуляторы, способные обрабатывать данные самых разных типов, кажутся выдумкой, но это именно то, что вы получаете в виде языка программирования высокого уровня. Python поддерживает все арифметические операции, имеет модуль `math` и множество сторонних математических библиотек, позволяющих расширять среду программирования по мере необходимости. Поскольку Python является *полным по Тьюрингу*, с его помощью можно (в принципе) вычислить все, что может быть вычислено. Вам нужен только достаточно мощный компьютер, достаточно умная реализация или и то и другое.

В этой книге мы реализуем каждую новую математическую концепцию в виде кода на Python, пригодного для повторного использования. Самостоятельная работа станет для вас отличным способом закрепить понимание новых концепций и добавить в личную библиотеку новый инструмент. Проделав весь путь самостоятельно, вы сможете заменить отточенную популярную библиотеку, если захотите. В любом случае новые инструменты, созданные или заимствованные вами, позволят заложить основу для реализации еще более масштабных идей.

1.3.3. Создание абстракций с помощью функций

Процесс, который я только что описал, в программировании называется *абстракцией*. Например, чтобы не повторять утомительный счет по порядку, вы используете абстракцию сложения. Чтобы не повторять сложение, применяете абстракцию умножения и т. д.

Из всех способов создания абстракций в программировании наиболее важна для математики *функция*. Функция в Python — это способ повторного решения некоторой задачи. Функция может принимать входные данные и производить выходные данные. Например,

```
def greet(name):
    print("Hello %s!" % name)
```

позволяет поприветствовать нескольких человек с помощью короткого и выразительного кода:

```
>>> for name in ["John","Paul","George","Ringo"]:
...     greet(name)
...
Hello John!
Hello Paul!
Hello George!
Hello Ringo!
```

Эта функция несет определенную пользу, но она не похожа на математическую функцию. Математические функции всегда принимают входные значения, всегда возвращают выходные значения и не имеют побочных эффектов.

Функции, которые ведут себя подобно математическим, в программировании называются *чистыми функциями*. Например, квадратичная функция $f(x) = x^2$ принимает число и возвращает произведение числа на самого себя. Вычисляя $f(3)$, вы получаете в результате 9. Это не означает, что число 3 теперь изменилось и превратилось в 9. Это означает, что 9 является соответствующим выходом для входа 3 в функции f. Функцию возведения в квадрат можно представить как машину, которая получает числа через входное окно и выдает результаты (числа) через выходное окно (рис. 1.16).

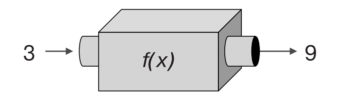


Рис. 1.16. Функция как машина с входным и выходным окнами

Это простая и полезная мысленная модель, и я буду возвращаться к ней на протяжении всей книги. Больше всего в ней мне нравится возможность представить функцию как объект. В математике, как и в Python, функции — это данные, которыми можно манипулировать независимо и которые можно даже передавать другим функциям.

Математика может казаться пугающей, потому что она абстрактна. Помните, что, как и в любой хорошо написанной программе, абстракции вводятся в рассуждения не просто так — они помогают организовывать и передавать более крупные и емкие положения. Когда вы поймете их и воплотите в код, перед вами откроются захватывающие возможности.

Я надеюсь, теперь вы поверите в то, что математика может применяться в разработке программного обеспечения во многих областях. У вас, как у программиста, уже сложился правильный образ мышления и имеются начальные знания, необходимые для изучения новых математических положений. Материал, представленный в этой книге, обогащает меня как в профессиональном, так и в личном плане, и я надеюсь, что обогатит и вас. Давайте начнем!

КРАТКИЕ ИТОГИ ГЛАВЫ

- Существуют интересные и перспективные области разработки программного обеспечения с применением математики.
- Математика может помочь количественно оценить тенденции, наблюдающиеся в данных и меняющиеся со временем, чтобы, например, предсказать изменение цен на акции.
- Разные типы функций передают разные виды качественного поведения. Например, функция экспоненциального уменьшения отражает динамику снижения стоимости подержанного автомобиля с каждой пройденной милей.
- Наборы чисел, называемые *векторами*, представляют многомерные данные. В частности, трехмерные векторы, выражающиеся тройками чисел, могут представлять точки в пространстве. Манипулируя треугольниками, заданными векторами, можно создавать сложную трехмерную графику.
- *Математический анализ* — это область математики, занимающаяся исследованием непрерывных изменений, и многие законы физики записаны в терминах уравнений счисления, которые называют *дифференциальными уравнениями*.
- Трудно учить математику по традиционным учебникам! Математику следует учить, исследуя ее, а не просто знакомясь с определениями и теоремами.
- Как программист, вы умеете мыслить и общаться, используя точные определения. Этот навык поможет вам учить математику.

Часть I

Векторы и графика

В первой части этой книги мы углубимся в область математики, которая называется *линейной алгеброй*. Линейная алгебра — это раздел математики, связанный с вычислениями на многомерных данных. Понятие «размерность» — геометрическое, вы наверняка интуитивно понимаете, что я имею в виду, когда говорю, что квадрат — двухмерная фигура, а куб — трехмерная. Кроме всего прочего, линейная алгебра позволяет превратить геометрические идеи мерности в нечто, что можно вычислить конкретно.

Базовая концепция линейной алгебры — это *вектор*, который можно представить как точку в некотором многомерном пространстве. Например, вы наверняка слышали в школе о двухмерной координатной плоскости. Как будет говориться в главе 2, двухмерные векторы соответствуют точкам на плоскости и их можно представить упорядоченными парами чисел (x, y). В главе 3 мы рассмотрим трехмерное пространство, векторы (точки) в котором представлены тройками чисел (x, y, z). В обоих случаях можно использовать наборы векторов для определения геометрических фигур, из которых в свою очередь можно скомпоновать интересные графические изображения.

Еще одно ключевое понятие линейной алгебры — *линейное преобразование*, с которым мы познакомимся в главе 4. Линейное преобразование — это своего рода функция, которая принимает на входе вектор и возвращает на выходе вектор, сохраняя при этом геометрию (в особом смысле) задействованных векторов. Например, если набор векторов (точек) лежит на прямой в двухмерном пространстве, то после линейного преобразования они так и останутся на одной прямой. В главе 5 мы введем понятие *матриц* — прямоугольных массивов чисел, которые могут выражать линейные преобразования. Кульминацией изучения линейных преобразований станет их последовательное применение к трехмерной графике в программе на Python для создания анимационного эффекта.

Изобразить мы можем только двух- и трехмерные векторы и линейные преобразования, но можем определить векторы с любым количеством измерений. В n-мерном пространстве вектор определяется упорядоченным набором из n чисел $(x_1, x_2, ..., x_n)$. В главе 6 мы воспользуемся концепциями двух- и трехмерного

пространства, чтобы определить обобщенную идею *векторного пространства* и более конкретное понятие *размерности*. В частности, увидим, что можно рассматривать цифровые изображения, состоящие из пикселов, как векторы в многомерном векторном пространстве и манипулировать этими изображениями с помощью линейных преобразований.

Наконец, в главе 7 исследуем самый распространенный вычислительный инструмент линейной алгебры — решение *систем линейных уравнений*. Как вы, возможно, помните из школьного курса алгебры, решение двух линейных уравнений с двумя переменными x и y сообщает нам координаты точки пересечения двух линий на плоскости. В общем случае решение системы линейных уравнений сообщает координаты пересечения линий, плоскостей и обобщений более высокой мерности в векторном пространстве. Имея возможность автоматически решать такие задачи на Python, мы будем использовать ее для создания первой версии движка видеоигры.

2 Рисование с помощью двухмерных векторов

В этой главе

- ✓ Создание двухмерных фигур, представленных наборами векторов, и управление ими.
- ✓ Представление двухмерных векторов в виде стрелок, точек и упорядоченных пар координат.
- ✓ Использование векторной арифметики для преобразования фигур на плоскости.
- ✓ Применение тригонометрии для измерения расстояний и углов на плоскости.

Вероятно, у вас уже есть некоторое представление о том, что такое двухмерный или трехмерный объект. *Двухмерный* объект — плоский, как изображение на листе бумаги или на экране компьютера. Он имеет только два размера — высоту и ширину. *Трехмерный* объект — это объект в нашем физическом мире, он имеет не только высоту и ширину, но и глубину.

Модели двух- и трехмерных объектов играют важную роль в программировании. Все, что отображается на экране телефона, планшета или персонального компьютера, — это двухмерное изображение, имеющее определенную ширину и высоту, измеряемую в пикселах. Любая имитация, игра или анимация, представляющая физический мир, сохраняет объекты в трехмерном виде и проецирует их на

двухмерную плоскость экрана. В приложениях виртуальной и дополненной реальности трехмерные модели должны сочетаться с реальными трехмерными данными о положении пользователя и перспективе.

Наш повседневный опыт неразрывно связан с тремя измерениями, но некоторые данные удобнее представлять как имеющие большее число измерений. В физике принято рассматривать время как четвертое измерение. Если физический объект существует в некоторой точке в трехмерном пространстве, то событие происходит в некоторой точке в трехмерном пространстве в определенный момент времени. В задачах науки о данных наборы данных обычно имеют гораздо больше измерений. Например, пользователь веб-сайта может характеризоваться сотнями атрибутов, описывающих модель его поведения. Для решения задач с многомерными данными в графике, физике и анализе данных необходима основа, позволяющая работать с многомерными данными. Такой основой является векторная математика.

Векторы — это объекты, находящиеся в многомерных пространствах. Для них существуют свои представления об арифметике (сложение, умножение и т. д.). Для начала познакомимся с двухмерными векторами, которые легко визуализируются и вычисляются. В этой книге мы часто будем прибегать к двухмерным векторам и используем их в качестве мысленной модели при рассуждениях о многомерных задачах.

2.1. ИЗОБРАЖЕНИЕ ДВУХМЕРНЫХ ВЕКТОРОВ

Двухмерный мир плоский, как лист бумаги или экран компьютера. На языке математики плоское двухмерное пространство называется *плоскостью*. Объект в двухмерном мире имеет два измерения — высоту и ширину, но не имеет третьего измерения — глубины. Точно так же двумя элементами информации — расстоянием по вертикали и по горизонтали — можно описать местоположение в двухмерном мире. Для описания местоположения точек на плоскости нужна опорная точка. Мы называем ее *началом координат*. Эта взаимосвязь показана на рис. 2.1.

На выбор имеется множество точек, но мы должны зафиксировать одну из них в качестве начала координат. Чтобы отличить ее от других, отметим начало координат крестиком (×), а не точкой, как показано на рис. 2.1. От начала координат мы можем нарисовать стрелку (как на рис. 2.1), чтобы показать относительное местоположение другой точки.

Двухмерный вектор — это точка на плоскости относительно начала координат. Точно так же вектор можно представить как прямую стрелку на плоскости; любую стрелку можно нарисовать так, что она будет начинаться в начале координат и указывать на конкретную точку (рис. 2.2).

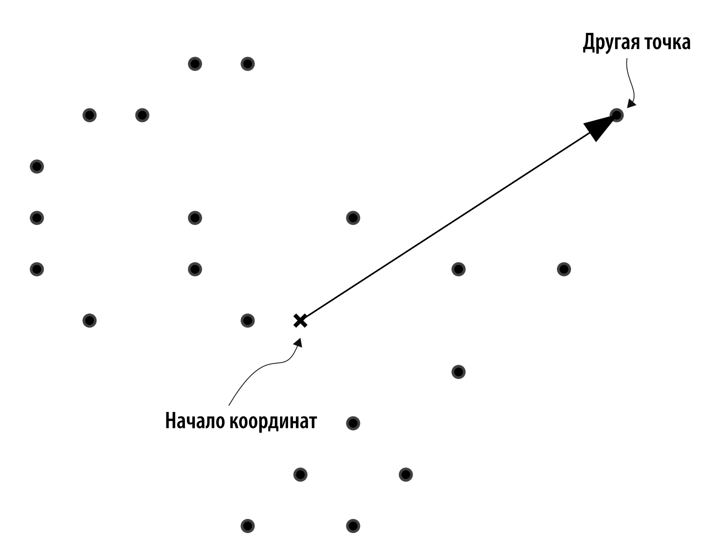


Рис. 2.1. Местоположение одной из множества точек на плоскости относительно начала координат



Рис. 2.2. Стрелка на плоскости указывает на точку относительно начала координат

Для представления векторов в этой и следующих главах будем использовать и стрелки, и точки. С точками работать удобнее, потому что из них можно построить интересные рисунки. Если соединить точки, как показано на рис. 2.3, то получится динозавр.

Когда компьютер отображает двух- или трехмерный рисунок, начиная с моего скромного динозавра и заканчивая полнометражным мультфильмом Pixar, он руководствуется точками, или векторами, соединенными отрезками, формирующими желаемые фигуры. Чтобы создать рисунок, нужно выбрать векторы в нужных местах, что требует тщательного их измерения. Посмотрим, как измерять векторы на плоскости.

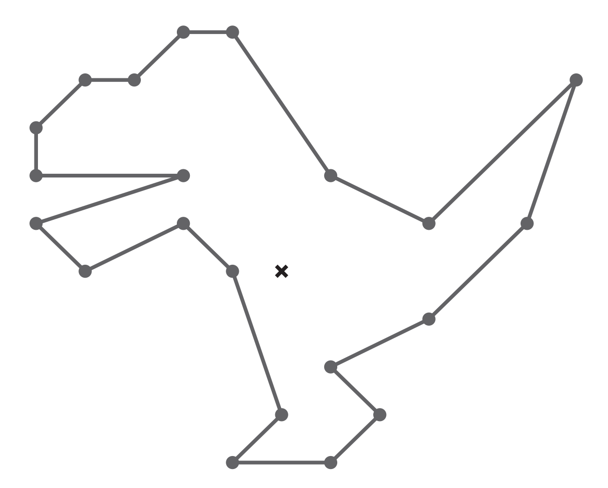


Рис. 2.3. Если соединить точки на плоскости, получится фигура

2.1.1. Представление двухмерных векторов

С помощью линейки мы можем измерить один параметр, например длину объекта. Чтобы выполнить измерения в двухмерном пространстве, нужны две линейки. Эти линейки называются *осями*. Мы должны расположить их на плоскости перпендикулярно друг другу так, чтобы они пересекались в начале координат. На рис. 2.4 с нанесенными осями видно, что у нашего динозавра есть верх и низ, а также лево и право. Горизонтальная ось называется *осью x*, а вертикальная — *осью y*.

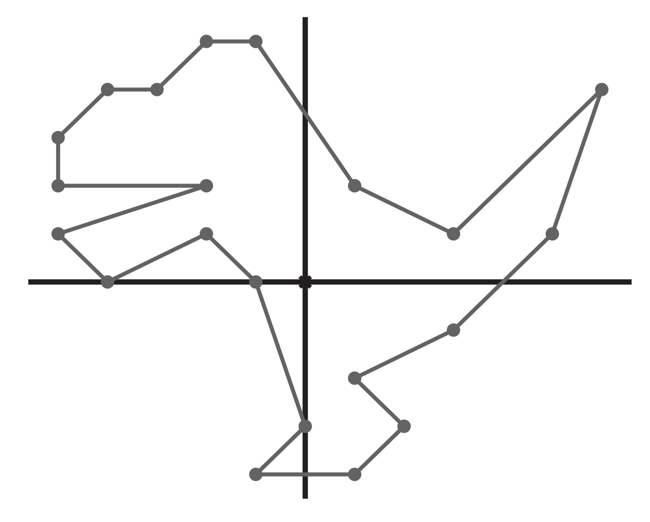


Рис. 2.4. Динозавр, нарисованный на плоскости с осями *x* и *y*

Ориентируясь по осям, можно сказать, например: «Четыре точки находятся выше и правее начала координат». Но нам нужны более точные количественные меры. На линейке есть деления, по которым можно определить, сколько единиц отмерено. Точно так же мы можем добавить на наш двухмерный чертеж линии координатной сетки, перпендикулярные осям, которые помогут точно определить относительное положение точек. По соглашению начало координат находится на отметке 0 по обеим осям, x и y (рис. 2.5).

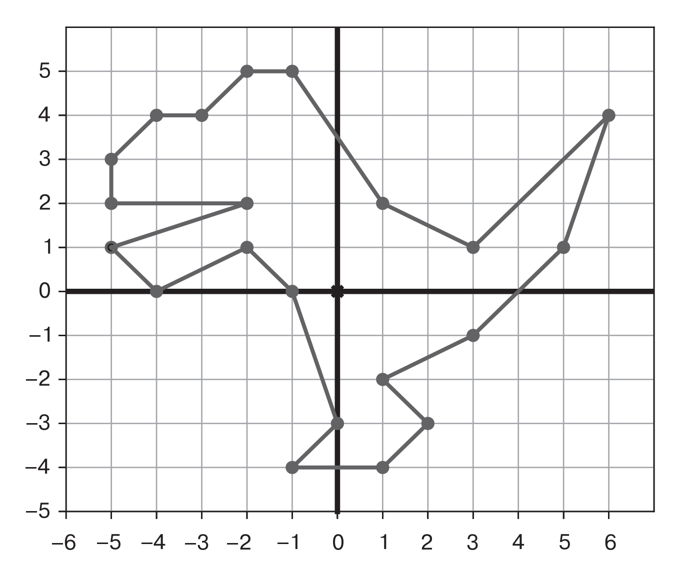


Рис. 2.5. Координатная сетка помогает определить местоположение точек относительно осей

Используя эту сетку, мы можем измерять векторы на плоскости. Например, на рис. 2.5 кончик хвоста динозавра совпадает с положительным значением 6 по оси x и положительным значением 4 по оси y. Мы могли бы объявить, что эти расстояния измеряются в сантиметрах, дюймах, пикселах или любых других единицах, но обычно единицы измерения оставляют неопределенными, если не имеется в виду конкретный случай.

Числа 6 и 4 называются *координатой x* и *координатой y* точки соответственно, и их достаточно, чтобы точно сказать, о какой точке идет речь. Обычно координаты записываются в виде *упорядоченной пары* (или *кортежа*), в которой координата x стоит первой, а координата y — второй, например (6, 4). На рис. 2.6 показано, как теперь можно описать один и тот же вектор тремя способами.

По другой паре координат, например (–3, 4,5), можно найти представляющую их точку, или стрелку на плоскости. Чтобы добраться до точки на плоскости с этими координатами, встаньте в начало координат, выполните три шага по линиям сетки влево (потому что координата $x = -3$), а затем четыре с половиной шага

по линиям сетки вверх (потому что координата $y = 4{,}5$). Точка не будет лежать на пересечении двух линий сетки, но это нормально — любая пара действительных чисел дает нам некоторую точку на плоскости. Соответствующая стрелка будет изображать прямой путь от начала координат к этой точке и указывать вверх и влево (или на северо-запад, если хотите). Попробуйте нарисовать этот рисунок для практики!

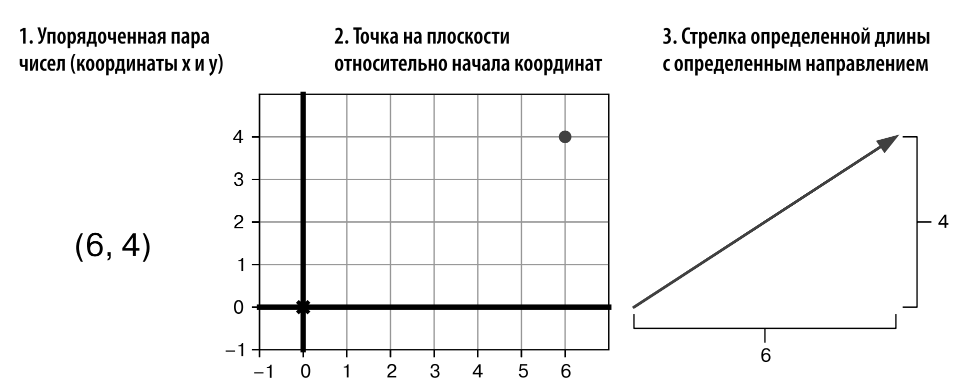


Рис. 2.6. Три способа описания одного и того же вектора

2.1.2. Рисование двухмерных изображений на Python

Создавая изображение на экране, вы работаете в двухмерном пространстве. Пикселы на экране — это доступные точки на плоскости. Но их координаты выражаются целыми числами, а не действительными, и нет возможности включить подсветку пространства между пикселами. Однако большинство графических библиотек позволяют работать с координатами в виде действительных чисел и автоматически преобразуют их в целочисленные координаты пикселов на экране.

У нас на выбор есть множество языков и библиотек для определения и отображения графики на экране: OpenGL, CSS, SVG и т. д. В Python, например, есть такие библиотеки, как Pillow и Turtle, которые хорошо подходят для создания рисунков на основе векторных данных. Для создания рисунков в этой главе я задействую небольшой набор собственных функций, использующих функции из библиотеки Matplotlib для Python. Это позволит нам сосредоточиться на применении Python для создания изображений из векторных данных. Поняв суть процесса, вы сможете выбрать любую другую библиотеку.

Самая важная функция называется `draw`. Она принимает данные, представляющие геометрические объекты, и именованные аргументы, определяющие некоторые характеристики рисунка. В табл. 2.1 перечислены классы Python, представляющие разные типы геометрических объектов, которые можно нарисовать.

Таблица 2.1. Некоторые классы Python, представляющие геометрические фигуры, которые можно использовать с функцией draw

Класс	Пример конструктора	Описание
Polygon	Polygon(*vectors)	Рисует многоугольник, вершины которого заданы списком векторов vectors
Points	Points(*vectors)	Представляет список точек для рисования, по одной для каждого вектора в списке vectors
Arrow	Arrow(tip) Arrow(tip, tail)	Рисует стрелку от начала координат до вектора tip или от вектора tail до вектора tip, если задан аргумент tail
Segment	Segment(start,end)	Рисует отрезок, соединяющий векторы start и end

Вы можете найти эти функции в файле с исходным кодом `vector_drawing.py`. В конце главы я чуть больше расскажу об их реализациях.

ПРИМЕЧАНИЕ

Для этой главы (и всех последующих) в папке с исходным кодом имеется блокнот Jupyter, показывающий, как запускать (по порядку) примеры, представленные в главе, включая импорт функций из модуля vector_drawing. Если вы этого еще не сделали, то обратитесь к приложению А, где рассказывается, как настроить Python и Jupyter.

Используя эти функции рисования, можно нарисовать точки, очерчивающие контур динозавра (см. рис. 2.5):

```
from vector_drawing import *

    dino_vectors = [(6,4), (3,1), (1,2), (-1,5), (-2,5), (-3,4), (-4,4),
    # добавьте сюда 16 недостающих векторов
]

draw(
    Points(*dino_vectors)
)
```

Я не до конца заполнил список `dino_vectors`, но если вы добавите недостающие векторы, то этот код нарисует точки, как показано на рис. 2.7 (их расположение в точности соответствует точкам на рис. 2.5).

На следующем шаге мы можем соединить некоторые точки отрезками. Первый отрезок, например, соединяет точку (6, 4) с точкой (3, 1) на хвосте динозавра. Мы можем соединить эти точки отрезком с помощью функции:

```
draw(
    Points(*dino_vectors),
    Segment((6,4),(3,1))
)
```

и получить картину, изображенную на рис. 2.8.

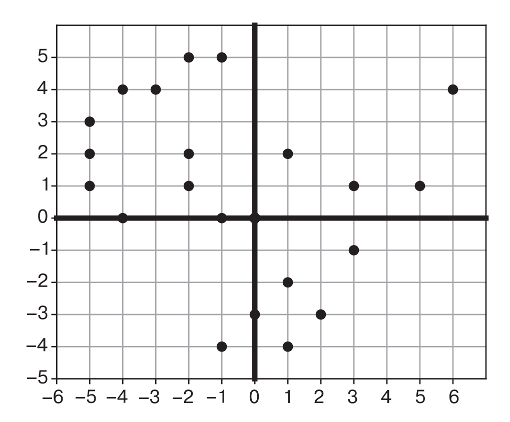


Рис. 2.7. Точки, образующие контур динозавра, нарисованные функцией draw

Рис. 2.8. Точки, образующие контур динозавра, и отрезок, соединяющий две первые точки в списке, (6, 4) и (3, 1)

Отрезок на самом деле представляет собой набор точек, включающий точки (6, 4) и (3, 1), а также все точки, лежащие на прямой между ними. Функция `draw` автоматически окрашивает все пикселы в этих точках в синий цвет. Класс `Segment` (отрезок) — удобная абстракция, потому что избавляет от необходимости строить из точек все отрезки, составляющие геометрическую фигуру (в данном случае динозавра). Нарисовав еще 20 отрезков, мы получим полный контур динозавра (рис. 2.9).

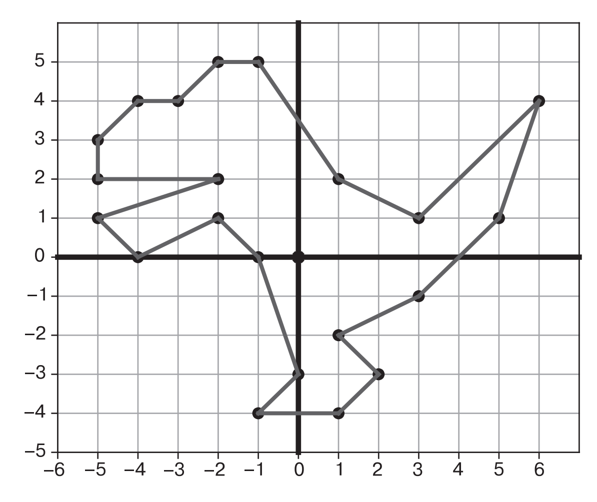


Рис. 2.9. 21 вызов функции даст нам 21 отрезок, образующий контур динозавра

Теперь мы можем нарисовать любую двухмерную фигуру, какую захотим, при условии, что у нас будут все векторы, определяющие ее. Придумывать все координаты вручную довольно утомительно, поэтому я предлагаю перейти к знакомству со способами вычислений для векторов, которые автоматически определяют их координаты.

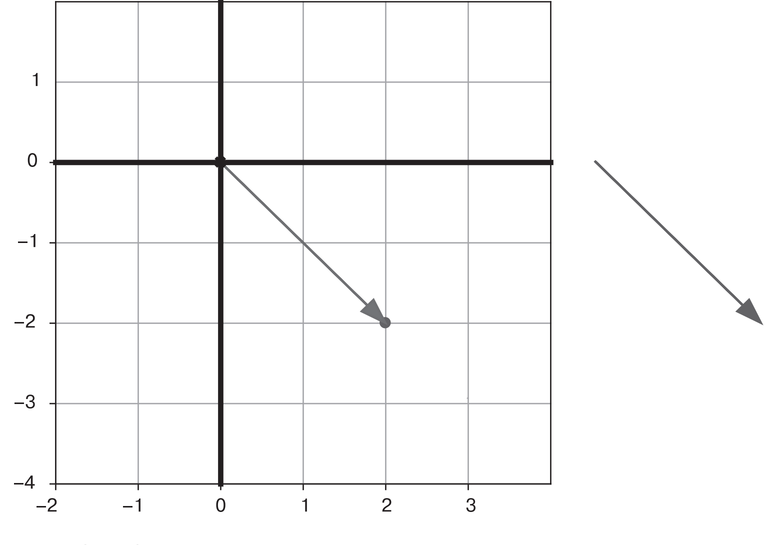
2.1.3. Упражнения

Упражнение 2.1. Какие координаты имеет точка на кончике пальца лапы динозавра?

Решение. $(-1, -4)$

Упражнение 2.2. Нарисуйте на плоскости точку $(2, -2)$ и стрелку к ней.

Решение. Результат с нарисованной точкой $(2, -2)$ и стрелкой к ней должен выглядеть так:



Точка (2, –2) и стрелка к ней

Упражнение 2.3. Глядя на расположение точек на рис. 2.7 (или рис. 2.5), определите векторы, не включенные в список `dino_vectors`. Например, я включил в него вектор (6, 4), определяющий точку кончика хвоста динозавра, но не включил точку (–5, 3) на его носу. По окончании список `dino_vectors` должен содержать 21 вектор в виде пар координат.

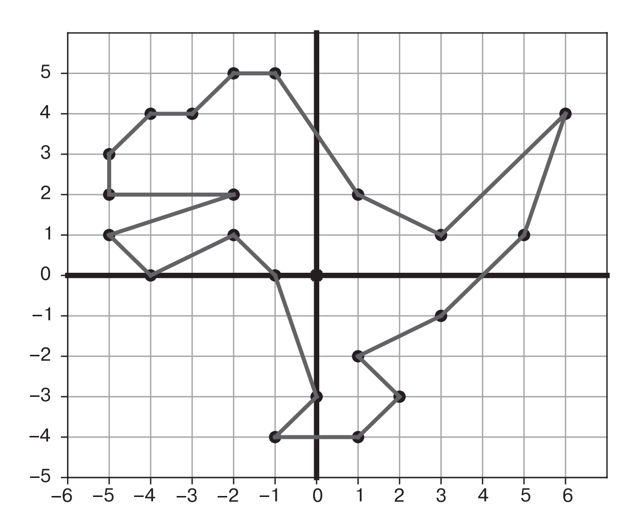
Решение. Вот как выглядит полный список векторов, определяющих контур динозавра:

```
dino_vectors = [(6,4), (3,1), (1,2), (-1,5), (-2,5), (-3,4), (-4,4),
                (-5,3), (-5,2), (-2,2), (-5,1), (-4,0), (-2,1), (-1,0),
                (0,-3), (-1,-4), (1,-4), (2,-3), (1,-2), (3,-1), (5,1)
               ]
```

Упражнение 2.4. Нарисуйте динозавра, создав объект `Polygon` со списком вершин `dino_vectors`.

Решение

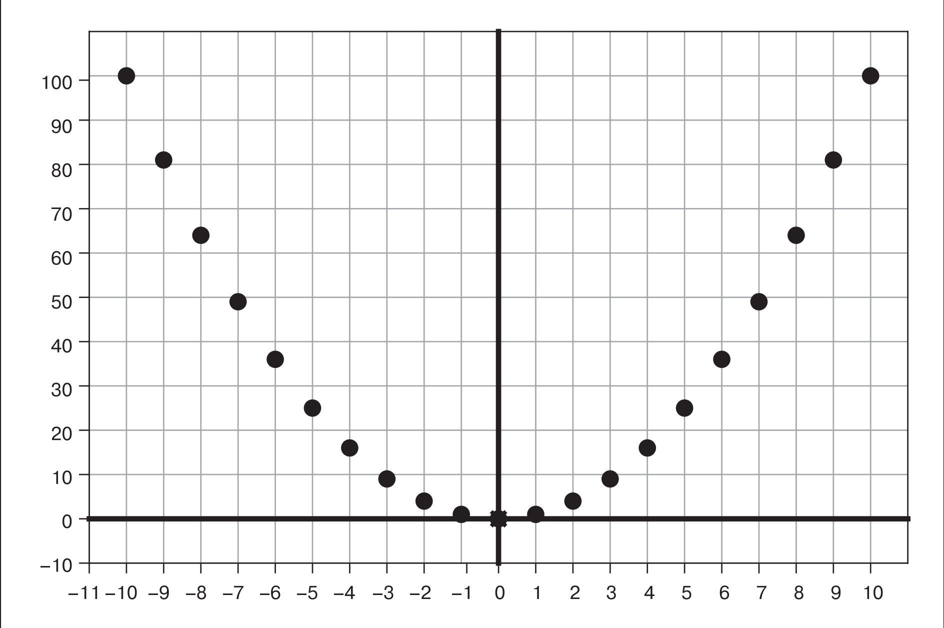
```
draw(
    Points(*dino_vectors),
    Polygon(*dino_vectors)
)
```



Динозавр, нарисованный в виде многоугольника (с помощью класса Polygon)

Упражнение 2.5. Нарисуйте векторы `(x, x**2)` для x в диапазоне от $x = -10$ до $x = 10$ в виде точек с помощью функции `draw`. Что получится в итоге?

Решение. Пары координат образуют график функции $y = x^2$ для целых чисел от −10 до 10:



Точки на графике функции $y = x^2$

Чтобы построить этот график, я использовал два именованных аргумента функции `draw`. Аргумент `grid=(1,10)` рисует вертикальные линии координатной сетки с шагом в 1 единицу и горизонтальные — с шагом 10 единиц. В именованном аргументе `nice_aspect_ratio` я передал значение `False`, сообщив функции `draw`, что нет необходимости сохранять одинаковые масштабы по осям x и y:

```
draw(
    Points(*[(x,x**2) for x in range(-10,11)]),
    grid=(1,10),
    nice_aspect_ratio=False
)
```

2.2. АРИФМЕТИКА ДВУХМЕРНЫХ ВЕКТОРОВ

Подобно числам, векторы имеют свою арифметику, мы можем комбинировать векторы и операции для получения новых векторов. Разница лишь в том, что векторы позволяют визуализировать результаты. Все операции векторной арифметики дают возможность выполнять не только алгебраические, но и геометрические преобразования. Начнем с самой простой операции — *сложения векторов*.

Сложение векторов выполняется просто: координаты x двух векторов складываются, и получается координата x их суммы, затем складываются координаты y, получается координата y суммы. В результате получается *векторная сумма*. Например, $(4, 3) + (-1, 1) = (3, 4)$, потому что $4 + (-1) = 3$ и $3 + 1 = 4$. Сложение векторов реализуется на Python одной строкой:

```
def add(v1,v2):
    return (v1[0] + v2[0], v1[1] + v2[1])
```

Поскольку векторы можно интерпретировать как стрелки или точки на плоскости, мы можем изобразить результат (рис. 2.10). Достичь точки $(-1, 1)$ на плоскости можно, начав с начала координат $(0, 0)$ и переместившись на одну единицу влево и на одну единицу вверх. Чтобы получить векторную сумму $(4, 3) + (-1, 1)$, нужно начать не с начала координат, а из точки $(4, 3)$ и переместиться на одну единицу влево и на одну единицу вверх. Проще говоря, нужно пройти сначала по одной стрелке, а потом по другой.

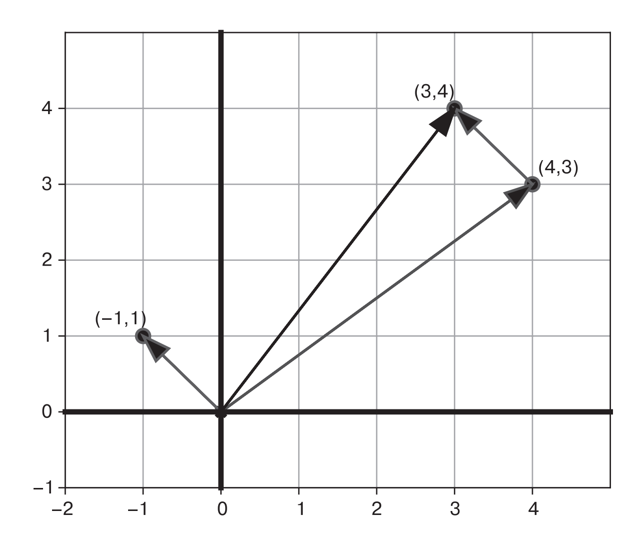


Рис. 2.10. Изображение суммы векторов (4, 3) и (–1, 1)

Правило сложения векторов как стрелок иногда называют *правилом треугольника*, потому что сумму можно получить, поместив начало второй стрелки в конец

первой (не изменяя ее длины и направления!) или, наоборот, поместив начало первой стрелки в конец второй (рис. 2.11).

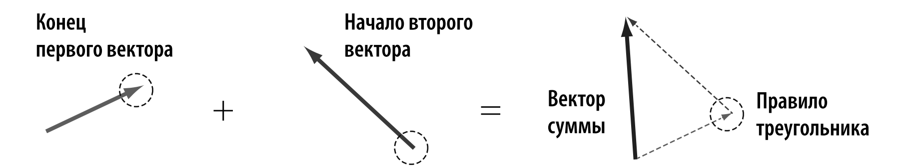


Рис. 2.11. Сложение векторов по правилу треугольника

Говоря о стрелках, я в действительности имею в виду определенное расстояние в определенном направлении. Если пройти одно расстояние в одном направлении и другое расстояние — в другом, то сумма векторов даст вам направление на конечную точку и расстояние до нее (рис. 2.12).

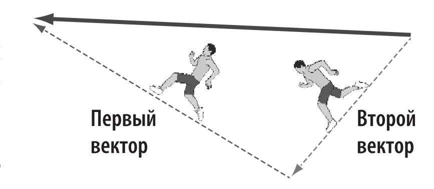


Рис. 2.12. Сумма векторов — это направление на конечную точку и расстояние до нее

Прибавление вектора производит эффект *параллельного переноса*, или *перемещения* существующей точки или набора точек. Если к каждому вектору в `dino_vectors` прибавить вектор (–1,5, –2,5), то получится новый список векторов, находящихся на 1,5 единицы левее и на 2,5 единицы ниже исходных векторов. Вот как это выглядит в коде:

```
dino_vectors2 = [add((-1.5,-2.5), v) for v in dino_vectors]
```

В результате получится тот же динозавр, смещенный вниз и влево на вектор (–1,5, –2,5). Чтобы увидеть это воочию (рис. 2.13), можно нарисовать обоих динозавров как многоугольники:

```
draw(
    Points(*dino_vectors, color=blue),
    Polygon(*dino_vectors, color=blue),
    Points(*dino_vectors2, color=red),
    Polygon(*dino_vectors2, color=red)
)
```

Стрелки на рис. 2.13, *справа*, показывают, что каждая точка переместилась вниз и влево на один и тот же вектор (–1,5, –2,5). Подобный перенос может пригодиться, например, если мы задумаем сделать динозавра движущимся персонажем в двухмерной компьютерной игре. В зависимости от кнопки, нажатой пользователем, динозавр мог бы перемещаться по экрану в том или ином направлении. Кстати, в главах 7 и 9 мы реализуем настоящую игру с подобной движущейся векторной графикой.

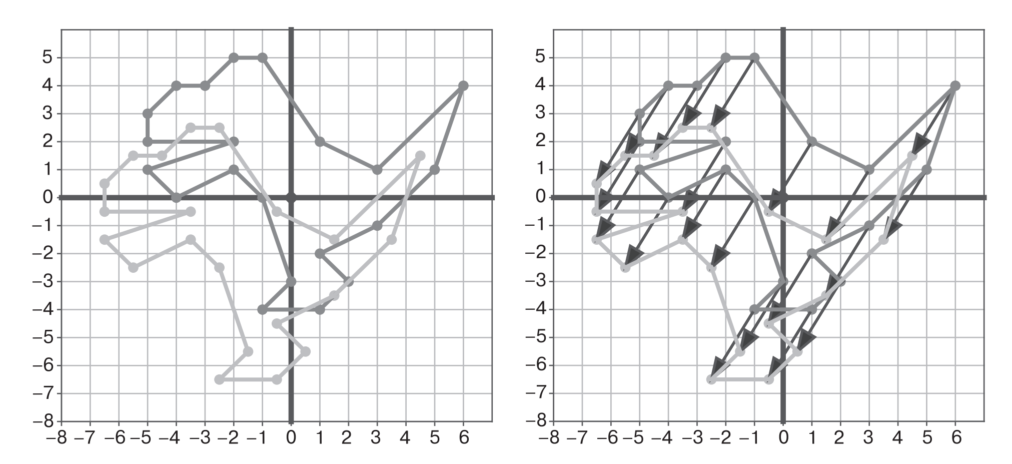


Рис. 2.13. Исходный динозавр *(темный)* и смещенная копия *(светлая)*. Каждая точка динозавра-копии смещена на вектор (–1,5, –2,5) вниз и влево

2.2.1. Компоненты вектора и его длина

Иногда бывает нужно разложить имеющийся вектор на сумму меньших векторов. Например, если бы я спросил, как пройти к некоторой точке в Нью-Йорке, то более подходящим для меня был бы ответ: «Пройдите четыре квартала на восток и три квартала на север», — а не: «Пройдите 800 метров на северо-восток». Точно так же иногда практичнее представлять векторы как суммы векторов, указывающих в направлениях x и y.

В качестве примера на рис. 2.14 показан вектор (4, 3), представленный в виде суммы (4, 0) + (0, 3). Сумма (4, 0) + (0, 3) приводит нас в ту же точку на плоскости, что и вектор (4, 3), но другим путем.

Два вектора — (4, 0) и (0, 3) — называются *компонентами* x и y соответственно. Не имея возможности ходить по диагонали на плоскости (представьте, что вы строите маршрут на карте Нью-Йорка), вы, чтобы добраться до точки назначения, должны будете пройти четыре единицы вправо, а затем три единицы вверх, то есть всего семь единиц.

Длина вектора — это длина стрелки, которая его представляет, или, что то же самое, расстояние от начала координат до точки, представляющей вектор. В Нью-Йорке это может быть расстояние между двумя перекрестками по прямой. Длину вектора в направлении x или y можно измерить непосредственно как количество отметок, пройденных вдоль соответствующей оси: оба вектора, (4, 0) или (0, 4), имеют одинаковую длину 4, хотя и указывают в разных направлениях. Однако в общем случае векторы могут располагаться по диагонали, и чтобы определить их длину, нужно выполнить некоторые вычисления.

Вспомните соответствующую формулу — *теорему Пифагора*, которая гласит, что квадрат длины наибольшей стороны прямоугольного треугольника (треугольника, две стороны которого сходятся под углом 90°) равен сумме квадратов длин меньших сторон. Наибольшая сторона называется *гипотенузой*, а ее длина выражается памятной формулой $a^2 + b^2 = c^2$, где a и b — длины меньших сторон. При $a = 4$ и $b = 3$ длину гипотенузы c можно получить, вычислив квадратный корень из $4^2 + 3^2$ (рис. 2.15).

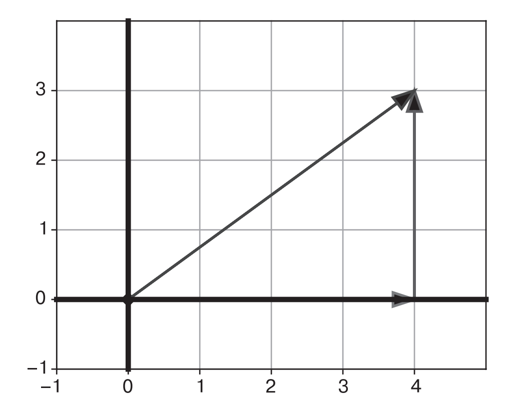


Рис 2.14. Представление вектора (4, 3) в виде суммы (4, 0) + (0, 3)

Рис. 2.15. Использование теоремы Пифагора для нахождения длины вектора по его *x*- и *y*-компонентам

Разложение вектора на компоненты всегда дает прямоугольный треугольник. Зная длины компонент, можно вычислить длину гипотенузы — длину вектора. Наш вектор (4, 3) эквивалентен сумме двух перпендикулярных векторов (4, 0) + (0, 3) со сторонами 4 и 3 соответственно. Длина вектора (4, 3) равна квадратному корню из $4^2 + 3^2$, то есть квадратному корню из 25, или 5. В городе с идеально квадратными кварталами прогулка на 4 квартала на восток и 3 квартала на север приведет нас в точку, находящуюся в 5 кварталах на северо-востоке.

Это редкий случай, когда расстояние получилось равным целому числу. Обычно длины, вычисляемые по теореме Пифагора, не являются целыми числами. Например, вот как длина вектора (–3, 7) определяется через длины его компонент 3 и 7:

$$\sqrt{3^2 + 7^2} = \sqrt{9 + 49} = \sqrt{58} = 7{,}61577\ldots$$

Эту формулу можно реализовать на Python в виде функции `length`, которая принимает двухмерный вектор и возвращает его длину в виде числа с плавающей точкой:

```
from math import sqrt
def length(v):
    return sqrt(v[0]**2 + v[1]**2)
```

2.2.2. Умножение вектора на число

Многократное прибавление вектора к самому себе интерпретируется однозначно как многократное добавление начала следующей стрелки в конец предыдущей по правилу треугольника. Пусть вектор **v** имеет координаты (2, 1), тогда пятикратное сложение **v** + **v** + **v** + **v** + **v** будет выглядеть так, как показано на рис. 2.16.

Если бы **v** был числом, мы бы не стали писать **v** + **v** + **v** + **v** + **v**, а использовали бы простое произведение 5**v**. Однако нет никаких причин, почему то же самое нельзя сделать с векторами. Результат пятикратного прибавления вектора **v** к самому себе — это вектор, указывающий в том же направлении, но имеющий длину в пять раз больше. Мы можем использовать это определение, чтобы умножать вектор на любое целое или дробное число.

Операция умножения вектора на число называется *скалярным умножением* или *скалярным произведением*. При работе с векторами обычные числа часто называют *скалярами*. Результат этой операции — *масштабирование* целевого вектора с заданным коэффициентом (скаляром). Неважно, является скаляр целым числом или нет, например, мы легко можем нарисовать вектор, который в 2,5 раза длиннее другого (рис. 2.17).

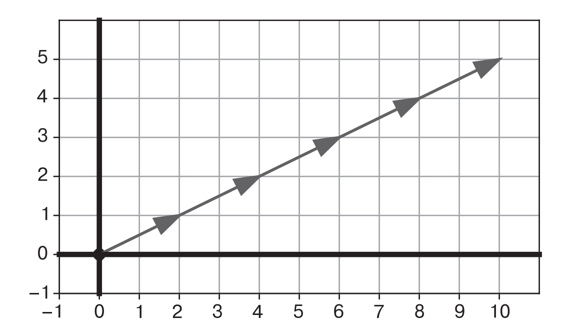


Рис. 2.16. Многократное прибавление вектора к самому себе

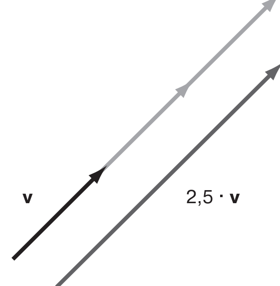


Рис. 2.17. Скалярное умножение вектора **v** на 2,5

Результат умножения в терминах векторных компонент — это масштабирование каждого компонента на один и тот же коэффициент. Скалярное умножение можно представить как изменение размера прямоугольного треугольника, заданного вектором и его компонентами, не влияющее на соотношение его сторон. На рис. 2.18 показаны вектор **v** и его скалярное произведение 1,5**v**. Скалярное произведение в 1,5 раза длиннее, и его компоненты тоже в 1,5 раза длиннее компонент исходного вектора **v**.

В координатах скалярное умножение вектора **v** = (6, 4) на 1,5 дает новый вектор (9, 6), где каждая компонента в 1,5 раза больше исходного значения. С точки зрения вычислений любое скалярное умножение вектора производится

умножением каждой координаты вектора на скаляр. В качестве второго примера рассмотрим умножение вектора **w** = (1,2, −3,1) на коэффициент 6,5, которое можно выполнить так:

$$6{,}5\mathbf{w} = 6{,}5 \cdot (1{,}2,\ -3{,}1) = (6{,}5 \cdot 1{,}2,\ 6{,}5 \cdot (-3{,}1)) = (7{,}8,\ -20{,}15).$$

Мы убедились, что это правило верно для дробных скаляров, но посмотрим, верно ли оно для отрицательных чисел. Если первоначальный вектор равен (6, 4), то как будет выглядеть результат его умножения на −1/2? Согласно правилу мы ожидаем, что в результате получится вектор (−3, −2). На рис. 2.19 показано, что этот вектор вдвое короче исходного и указывает в противоположном направлении.

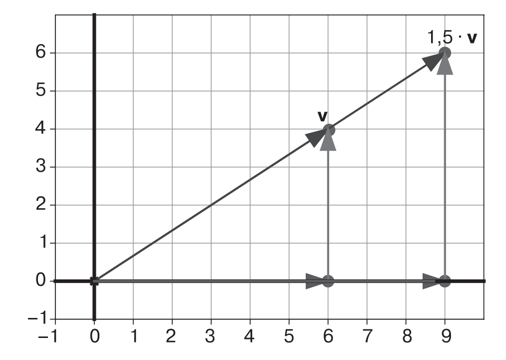


Рис. 2.18. Умножение вектора на скаляр приводит к масштабированию обеих компонент на тот же коэффициент

Рис. 2.19. Скалярное умножение вектора на отрицательное число −1/2

2.2.3. Вычитание, смещение и расстояние

Скалярное умножение согласуется с нашим пониманием умножения чисел. Умножение некоторого числа на целое число равноценно многократному сложению этого числа с самим собой данное количество раз, то же верно для векторов. Это же рассуждение можно применить для определения противоположных векторов и векторного вычитания.

Вектор −**v**, *противоположный* вектору **v**, совпадает с результатом скалярного произведения −1**v**. Если **v** определяется координатами (−4, 3), то противоположный ему вектор −**v** будет определяться координатами (4, −3), как показано на рис. 2.20. Этот результат получается умножением каждой координаты на −1, или, иначе говоря, сменой знака.

На числовой прямой есть только два направления в сторону от нуля — положительное и отрицательное. На плоскости множество направлений (на самом деле их бесконечно много), поэтому нельзя сказать, что какой-то из векторов, **v** и −**v**,

положительный, а другой отрицательный. Зато можно сказать, что вектор **–v**, противоположный любому данному вектору **v**, имеет ту же длину, но указывает в противоположном направлении.

Получив представление о противоположности векторов, можно определить операцию *векторного вычитания*. Для чисел результат вычитания $x - y$ совпадает с результатом сложения $x + (-y)$. То же верно и для векторов. Чтобы вычесть вектор **w** из вектора **v**, нужно прибавить вектор **–w** к **v**. Если рассматривать векторы **v** и **w** как точки, то разность **v** – **w** — это позиция **v** относительно **w**. Если рассматривать **v** и **w** как стрелки, выходящие из начала координат, то **v** – **w** — это стрелка, начинающаяся в точке **w** и заканчивающаяся в точке **v** (рис. 2.21).

Координаты разности **v** – **w** определяются как разность координат **v** и **w**. На рис. 2.21 изображены векторы **v** = (–1, 3) и **w** = (2, 2). Разность **v** – **w** имеет координаты (–1 – 2, 3 – 2) = (–3, 1).

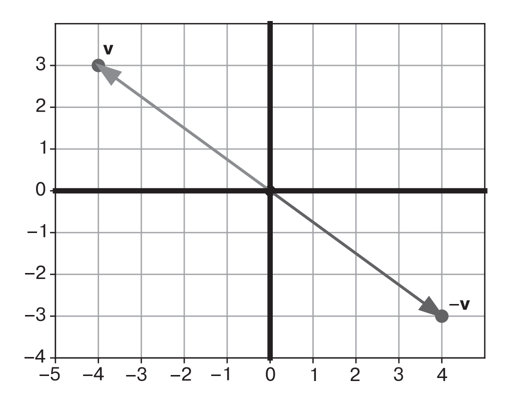


Рис. 2.20. Вектор **v** = (–4, 3) и противоположный ему вектор **–v** = (4, –3)

Рис. 2.21. Результат вычитания **v** – **w** — это стрелка из точки **w** в точку **v**

Посмотрим еще раз на разность векторов **v** = (–1, 3) и **w** = (2, 2). Вы можете воспользоваться функцией `draw`, чтобы нарисовать точки **v** и **w** и провести отрезок между ними. Вот как это выглядит в коде:

```
draw(
    Points((2,2), (-1,3)),
    Segment((2,2), (-1,3), color=red)
)
```

Разность векторов **v** – **w** = (–3, 1) сообщает, что если выйти из точки **w** и сделать три шага влево и один шаг вверх, то мы попадем в точку **v**. Иногда этот вектор называют *смещением* из **w** в **v**. Отрезок, соединяющий **w** и **v** на рис. 2.22 и нарисованный этим кодом на Python, определяет *расстояние* между двумя точками.

Длина отрезка вычисляется по теореме Пифагора:

$$\sqrt{(-3)^2+1^2}=\sqrt{9+1}=\sqrt{10}=3{,}162\ldots$$

Если смещение является вектором, то расстояние — это скаляр (одно число). Одного расстояния недостаточно, чтобы указать, как добраться из **w** в **v**, потому что существует множество точек, находящихся на одном и том же расстоянии от **w**. На рис. 2.23 показаны некоторые из этих точек, имеющие целочисленные координаты.

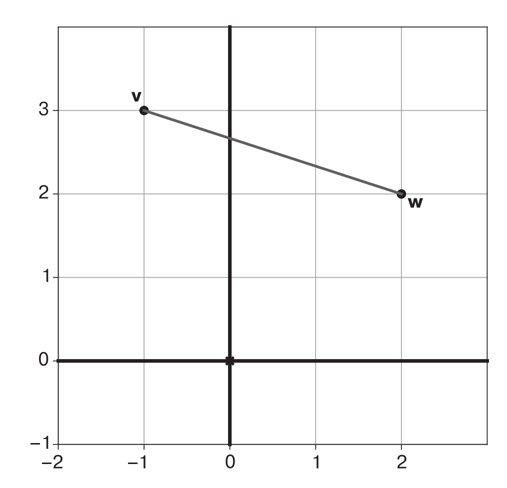


Рис. 2.22. Расстояние между двумя точками на плоскости

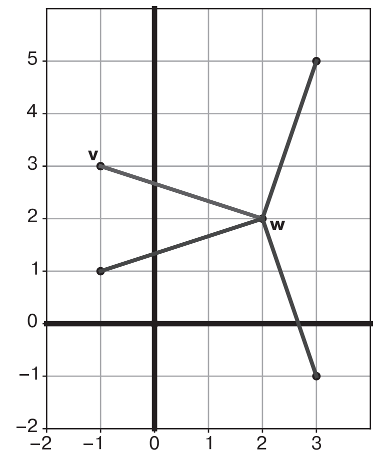


Рис. 2.23. Несколько точек, равноудаленных от **w** = (2, 2)

2.2.4. Упражнения

Упражнение 2.6. Даны векторы **u** = (–2, 0), **v** = (1,5, 1,5) и **w** = (4, 1). Определите результаты **u** + **v**, **v** + **w** и **u** + **w**. Определите результат **u** + **v** + **w**.

Решение. Для векторов **u** = (–2, 0), **v** = (1,5, 1,5) и **w** = (4, 1) результаты будут следующие:

u + **v** = (–0,5, 1,5);

v + **w** = (5,5, 2,5);

u + **w** = (2, 1);

u + **v** + **w** = (3,5, 2,5).

Упражнение 2.7. Мини-проект. Любое количество векторов можно сложить, просуммировав *все* координаты x и *все* координаты y. Например, сумма четырех векторов (1, 2) + (2, 4) + (3, 6) + (4, 8) имеет компоненту $x = 1 + 2 + 3 + 4 = 10$ и компоненту $y = 2 + 4 + 6 + 8 = 20$, что дает в результате вектор (10, 20). Реализуйте свою версию функции `add`, которая принимает любое количество векторов и находит их сумму.

Решение

```
def add(*vectors):
    return (sum([v[0] for v in vectors]), sum([v[1] for v in vectors]))
```

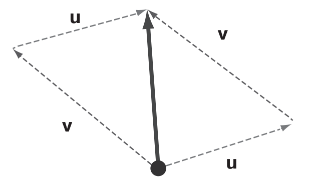
Упражнение 2.8. Напишите функцию `translate(translation, vectors)`, которая принимает вектор переноса `translation` и список входных векторов `vectors` и возвращает список векторов, полученных переносом исходных векторов на вектор `translation`. Например, вызов `translate((1,1), [(0,0), (0,1,), (-3, -3)])` должен вернуть `[(1,1), (1,2), (-2, -2)]`.

Решение

```
def translate(translation, vectors):
    return [add(translation, v) for v in vectors]
```

Упражнение 2.9. Мини-проект. Любая операция сложения векторов **v** + **w** дает тот же результат, что и **w** + **v**. Объясните, почему верно это утверждение, используя определение векторной суммы по координатам. Кроме того, нарисуйте схему, показывающую верность этого утверждения с геометрической точки зрения.

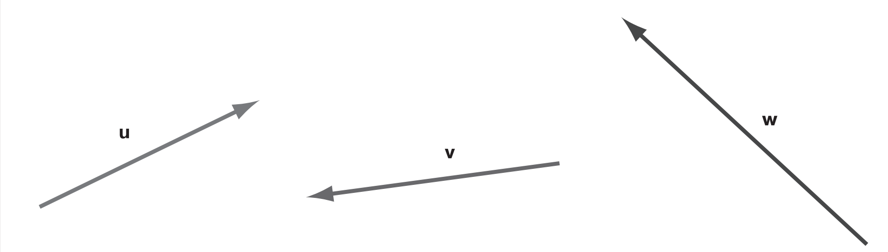
Решение. Если сложить два вектора, **u** = (a, b) и **v** = (c, d), координаты a, b, c и d которых являются действительными числами, то результатом будет вектор **u** + **v** = ($a + c$, $b + d$). Результатом сложения **v** + **u** будет ($c + a$, $d + b$) — та же пара координат, потому что от перемены мест действительных слагаемых сумма не меняется. Сложение векторов в любом порядке по правилу треугольника дает в результате один и тот же вектор. Визуально это можно изобразить, как показано на рисунке:



Сложение векторов в любом порядке по правилу треугольника дает в результате один и тот же вектор

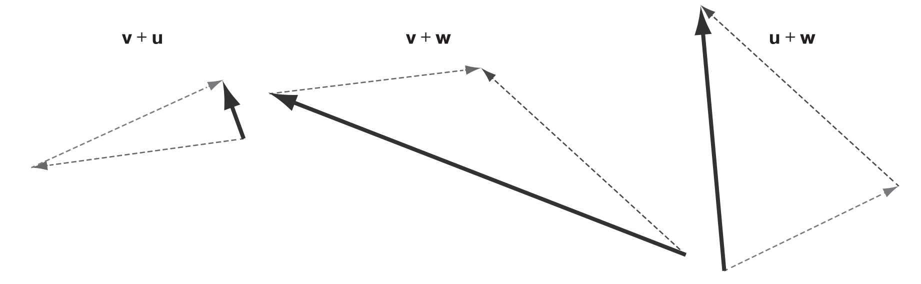
Независимо от порядка сложения векторов, **u** + **v** или **v** + **u** (изображены пунктирными линиями), всегда получается один и тот же вектор результата (изображен сплошной линией). С геометрической точки зрения векторы **u** и **v** определяют параллелограмм, а их сумма — диагональ этого параллелограмма.

Упражнение 2.10. Даны три вектора-стрелки, подписанные **u**, **v** и **w**. Сумма каких двух из них даст в результате самую длинную стрелку? Сумма каких двух из них даст в результате самую короткую стрелку?



Сумма каких двух из этих векторов даст в результате самую длинную и самую короткую стрелку?

Решение. Мы можем измерить длину каждой суммы векторов, приложив векторы друг к другу по правилу треугольника:



Попарные суммы исходных векторов

Взглянув на результаты, видим, что **v** + **u** — самый короткий вектор (**u** и **v** направлены почти в противоположные стороны и практически нейтрализуют друг друга). Самый длинный вектор — это **v** + **w**.

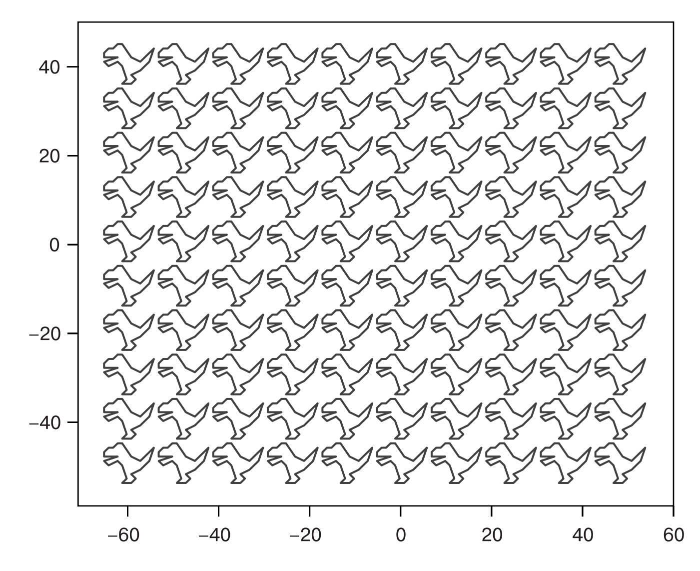
Упражнение 2.11. Мини-проект. Напишите функцию на Python, использующую сложение векторов, для одновременного отображения 100 непересекающихся копий динозавра. Это поможет вам воочию увидеть мощь компьютерной графики. Представьте, как утомительно было бы указывать все 2100 пар координат вручную!

Решение. Методом проб и ошибок можно подобрать такие векторы переноса по вертикали и горизонтали, чтобы получающиеся изображения динозавров не пересекались друг с другом, и установить соответствующие границы. Я решил не использовать линии сетки, оси, начало координат и точки, чтобы сделать рисунок более четким. Мой код выглядит так:

```
def hundred_dinos():
    translations = [(12*x,10*y)
                    for x in range(-5,5)
                    for y in range(-5,5)]
    dinos = [Polygon(*translate(t, dino_vectors),color=blue)
                for t in translations]
    draw(*dinos, grid=None, axes=None, origin=None)

hundred_dinos()
```

А так выглядит получившийся результат:



100 динозавров. Спасайся кто может!

Упражнение 2.12. Какая компонента, x или y, длиннее у вектора (3, –2) + + (1, 1) + (–2, –2)?

Решение. Результатом сложения векторов (3, –2) + (1, 1) + (–2, –2) является вектор (2, –3). Компонента x равна (2, 0), а компонента y — (0, –3). Компонента x имеет длину 2 единицы (вправо), а компонента y — 3 единицы (вниз, потому что она отрицательная). То есть компонента y длиннее.

Упражнение 2.13. Определите компоненты и длины векторов (–6, –6) и (5, –12).

Решение. Вектор (–6, –6) имеет компоненты (–6, 0) и (0, –6), обе длиной 6. Длина (–6, –6) равна квадратному корню из $6^2 + 6^2$, то есть примерно 8,485.

Вектор (5, –12) имеет компоненты (5, 0) и (0, –12) с длиной 5 и 12 соответственно. Длина (5, –12) равна квадратному корню из $5^2 + 12^2 = 25 + 144 = 169$, то есть 13.

Упражнение 2.14. Пусть есть вектор **v** длиной 6 с x-компонентой (1, 0). Определите возможные координаты **v**.

Решение. Компонента x (1, 0) имеет длину 1, а общая длина равна 6, поэтому длина b компоненты y должна удовлетворять уравнению $1^2 + b^2 = 6^2$, или $1 + b^2 = 36$. Отсюда следует, что $b^2 = 35$ и длина компоненты y приблизительно равна 5,916. Однако направление компоненты y по исходным данным определить нельзя. Вектор **v** может быть либо (1, 5,916), либо (1, –5,916).

Упражнение 2.15. Какой вектор в списке `dino_vectors` имеет наибольшую длину? Используйте функцию `length`, написанную ранее, чтобы быстро найти ответ.

Решение

```
>>> max(dino_vectors, key=length)
(6, 4)
```

Упражнение 2.16. Пусть есть вектор **w** с координатами $(\sqrt{2}, \sqrt{3})$. Вычислите примерные координаты вектора, получающегося в результате скалярного умножения $\pi\mathbf{w}$. Изобразите исходный и новый векторы.

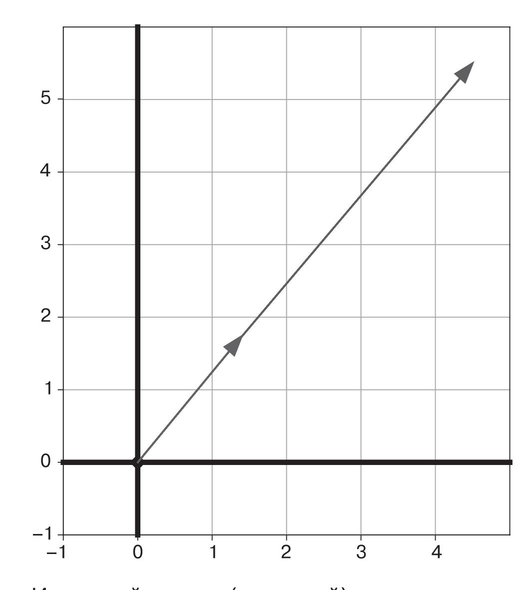
Решение. Значение $(\sqrt{2}, \sqrt{3})$ приблизительно равно:

$$(1{,}4142135623730951,\ 1{,}7320508075688772).$$

Умножив каждую координату на π, получаем:

$$(4{,}442882938158366,\ 5{,}441398092702653).$$

Исходный и получившийся вектор, который длиннее исходного, изображены далее:



Исходный вектор (короткий) и вектор, получившийся в результате умножения (длинный)

Упражнение 2.17. Напишите функцию `scale(s, v)` на Python, которая умножает вектор **v** на скаляр *s*.

Решение

```
def scale(scalar,v):
          return (scalar * v[0], scalar * v[1])
```

Упражнение 2.18. Мини-проект. Покажите алгебраически, что умножение координат на некоторый множитель увеличивает длину вектора на тот же множитель. Пусть вектор длиной *c* имеет координаты (a, b). Покажите, что для любого неотрицательного действительного числа *s* длина вектора (sa, sb) равна *sc*. (Это правило не относится к отрицательным значениям *s*, потому что вектор не может иметь отрицательную длину.)

Решение. Обозначим через $|(a, b)|$ длину вектора (a, b). Итак, согласно условию:

$$c = \sqrt{a^2 + b^2} = |(a, b)|.$$

Отсюда можно вычислить длину вектора (sa, sb):

$$|(sa, sb)| = \sqrt{(sa)^2 + (sb)^2} = \sqrt{s^2a^2 + s^2b^2} =$$
$$= \sqrt{s^2\left(a^2 + b^2\right)} = |s|\sqrt{a^2 + b^2} = |s|c.$$

Если *s* — неотрицательное число, то оно будет совпадать со своим абсолютным значением, $s = |s|$. Соответственно, длина вектора, полученного в результате умножения на скаляр *s*, будет равна *sc*, что и требовалось показать.

Упражнение 2.19. Мини-проект. Пусть $\mathbf{u} = (-1, 1)$, $\mathbf{v} = (1, 1)$, *r* и *s* — действительные числа. Предположим, что $-3 < r < 3$ и $-1 < s < 1$. Определите возможные точки на плоскости, где может оказаться вектор $r\mathbf{u} + s\mathbf{v}$.

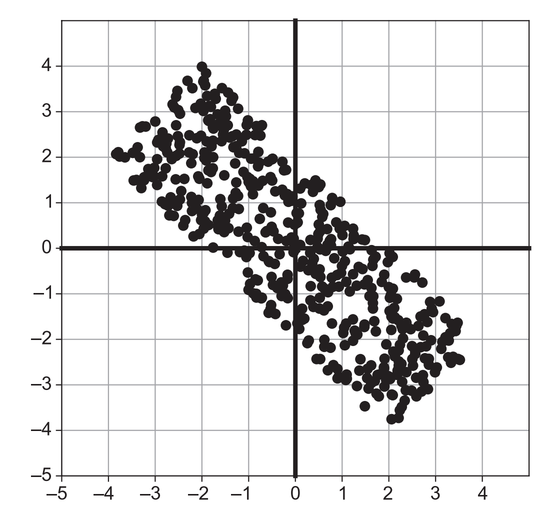
Обратите внимание на то, что операции с векторами выполняются в том же порядке, что и с числами, то есть сначала скалярное умножение, а затем сложение векторов (если скобки явно не определяют другой порядок).

Решение. Если $r = 0$, то возможные точки лежат на отрезке прямой от $(-1, -1)$ до $(1, 1)$. Если $r \neq 0$, то точки могут отстоять от этого отрезка в направлении $(-1, 1)$ или $-(-1, 1)$ не более чем на три единицы. Область расположения возможных точек ограничивается параллелограммом с вершинами $(2, 4)$, $(4, 2)$, $(2, -4)$ и $(4, -2)$. Мы можем проверить несколько случайных допустимых значений *r* и *s*, чтобы подтвердить это:

```
from random import uniform
u = (-1,1)
v = (1,1)
def random_r():
    return uniform(-3,3)
```

```
def random_s():
    return uniform(-1,1)
possibilities = [add(scale(random_r(), u), scale(random_s(), v))
                 for i in range(0,500)]
draw(
    Points(*possibilities)
)
```

Запустив этот код, вы получите картинку, как показано далее, на которой изображены точки, в которых может оказаться вектор $r\mathbf{u} + s\mathbf{v}$ с учетом установленных ограничений.



Точки, в которых может оказаться вектор $r\mathbf{u} + s\mathbf{v}$ с учетом заданных ограничений

Упражнение 2.20. Покажите алгебраически, почему вектор и его противоположность имеют одинаковую длину.

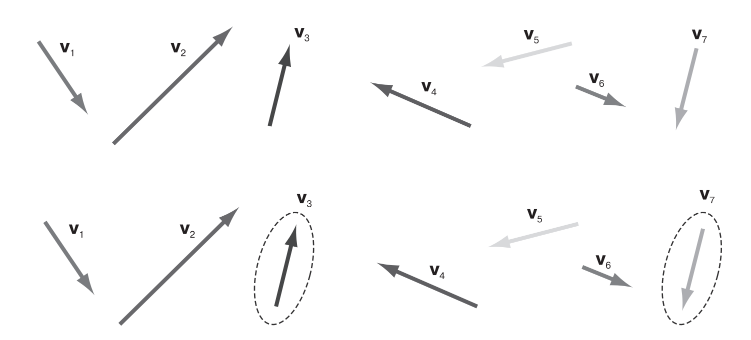
Подсказка. Подставьте координаты обоих векторов в теорему Пифагора.

Решение. Вектор, противоположный вектору (a, b), имеет координаты $(-a, -b)$, но это не влияет на длину:

$$\sqrt{(-a)^2 + (-b)^2} = \sqrt{(-a)(-a) + (-b)(-b)} = \sqrt{a^2 + b^2}.$$

Вектор $(-a, -b)$ имеет ту же длину, что и вектор (a, b).

Упражнение 2.21. Какие два из следующих семи векторов являются парой противоположных векторов?



Решение. Парой противоположных друг другу векторов являются $\mathbf{v}_3$ и $\mathbf{v}_7$.

Упражнение 2.22. Пусть **u** — некоторый двухмерный вектор. Определите координаты векторной суммы $\mathbf{u} + -\mathbf{u}$.

Решение. Двухмерный вектор **u** имеет некоторые координаты (a, b). Противоположный ему вектор имеет координаты $(-a, -b)$, поэтому

$$\mathbf{u} + (-\mathbf{u}) = (a, b) + (-a, -b) = (a - a, b - b) = (0, 0).$$

Ответ: (0, 0). Этот ответ имеет следующий геометрический смысл: если проследовать вдоль вектора, а затем вдоль его противоположности, то вы вернетесь в начало координат (0, 0).

Упражнение 2.23. Даны векторы $\mathbf{u} = (-2, 0)$, $\mathbf{v} = (1{,}5, 1{,}5)$ и $\mathbf{w} = (4, 1)$. Вычислите результаты следующих операций вычитания: $\mathbf{v} - \mathbf{w}$, $\mathbf{u} - \mathbf{v}$ и $\mathbf{w} - \mathbf{v}$.

Решение. Для $\mathbf{u} = (-2, 0)$, $\mathbf{v} = (1{,}5, 1{,}5)$ и $\mathbf{w} = (4, 1)$ получаем:

$\mathbf{v} - \mathbf{w} = (-2{,}5, 0{,}5)$;

$\mathbf{u} - \mathbf{v} = (-3{,}5, -1{,}5)$;

$\mathbf{w} - \mathbf{v} = (2{,}5, -0{,}5)$.

Упражнение 2.24. Напишите на Python функцию `subtract(v1, v2)`, которая принимает два двухмерных вектора и возвращает результат `v1 - v2`.

Решение.

```
def subtract(v1,v2):
    return (v1[0] - v2[0], v1[1] - v2[1])
```

Упражнение 2.25. Напишите на Python функцию `distance(v1, v2)`, которая возвращает *расстояние* между двумя векторами. (Обратите внимание на то, что функция `subtract` из предыдущего упражнения уже дает нужное *смещение*.)

Напишите на Python еще одну функцию, `perimeter(vectors)`, которая принимает список векторов и возвращает сумму расстояний от каждого предыдущего вектора до следующего и от последнего — до первого. Вычислите с ее помощью периметр динозавра, определяемого списком `dino_vectors`.

Решение. Расстояние — это просто длина разности двух входных векторов:

```
def distance(v1,v2):
    return length(subtract(v1,v2))
```

Чтобы вычислить периметр, нужно сложить расстояния между всеми парами смежных векторов в списке, а также между последним и первым векторами:

```
def perimeter(vectors):
    distances = [distance(vectors[i], vectors[(i+1)%len(vectors)])
                 for i in range(0,len(vectors))]
    return sum(distances)
```

Для проверки можно попробовать вычислить периметр квадрата с единичной стороной:

```
>>> perimeter([(1,0),(1,1),(0,1),(0,0)])
4.0
```

А затем вычислить периметр динозавра:

```
>>> perimeter(dino_vectors)
44.77115093694563
```

Упражнение 2.26. Мини-проект. Пусть есть вектор $\mathbf{u} = (1, -1)$ и существует другой вектор, $\mathbf{v}$, с положительными целыми координатами (n, m), такими, что $n > m$, находящийся на расстоянии 13 от $\mathbf{u}$. Определите смещение от $\mathbf{u}$ до $\mathbf{v}$.

Подсказка. Можете использовать Python для поиска вектора $\mathbf{v}$.

Решение. Для решения задачи нужно найти возможные целочисленные пары (n, m), где n находится в пределах 13 единиц от 1, а m — в пределах 13 единиц от -1:

```
for n in range(-12,15):
    for m in range(-14, 13):
        if distance((n,m), (1,-1)) == 13 and n > m > 0:
            print((n,m))
```

Такая пара только одна — (13, 4). Соответствующий ей вектор находится в 12 единицах правее и в 5 единицах выше вектора $(1, -1)$, то есть смещение равно (12, 5).

Длины вектора недостаточно для его описания, так же как недостаточно знать расстояние между векторами, чтобы перейти от одного к другому. В обоих случаях недостающий компонент — это *направление*. Зная длину и направление вектора, вы сможете найти его координаты. Фактически этим и занимается *тригонометрия*, и мы рассмотрим эту тему в следующем разделе.

2.3. УГЛЫ И ТРИГОНОМЕТРИЯ НА ПЛОСКОСТИ

До сих пор для измерения векторов на плоскости мы использовали лишь две «линейки» (оси x и y). Стрелка из начала координат имеет некоторое измеримое смещение в горизонтальном и вертикальном направлениях, и эти значения однозначно определяют вектор. Однако вместо двух линеек с тем же успехом можно было бы взять линейку и транспортир. Например, имея вектор (4, 3), можно измерить или вычислить его длину, равную 5 единицам, а затем с помощью транспортира определить направление, как показано на рис 2.24.

Этот вектор имеет длину 5 единиц и указывает в направлении примерно на 37° против часовой стрелки от положительного направления оси x. Эти измерения дают нам новую пару чисел (5, 37°), которые, подобно исходным координатам, однозначно определяют вектор. Эти числа называются *полярными координатами* и так же хорошо описывают точки на плоскости, как и числа, с которыми мы работали до сих пор и которые, кстати, называются *декартовыми координатами*.

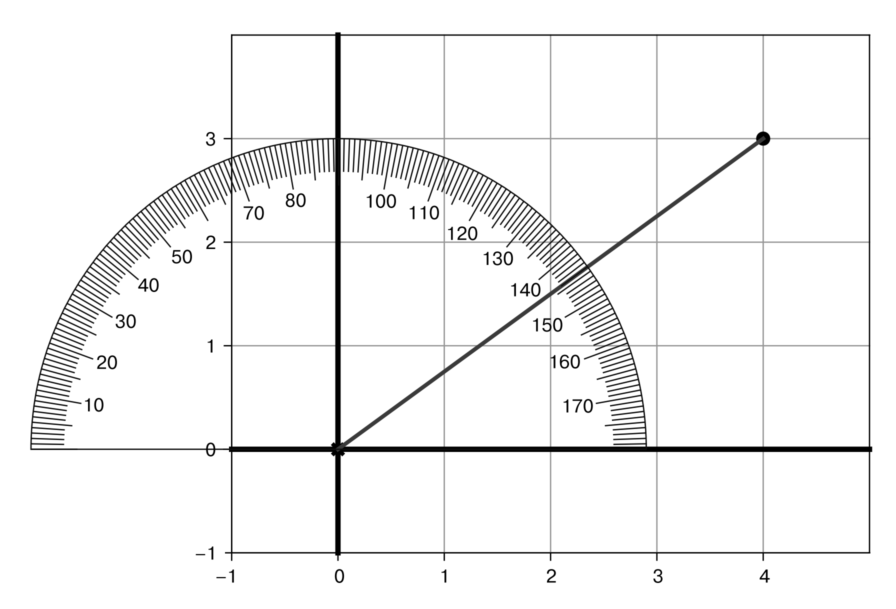


Рис. 2.24. Измерение угла между вектором и осями координат

В некоторых случаях, например при сложении векторов, проще оперировать декартовыми координатами. В других удобнее использовать полярные координаты, например, когда нужно определить вектор после поворота на некоторый угол. В программном коде у нас нет линеек или транспортиров, поэтому для преобразований между этими двумя системами координат применяются тригонометрические функции.

2.3.1. От углов к компонентам

Рассмотрим обратную задачу: представьте, что нам даны угол и расстояние, скажем, 116,57° и 3. Они определяют пару полярных координат (3, 116,57°). Как найти декартовы координаты для этого вектора геометрически?

Сначала можно наложить транспортир на начало координат, отметить правильное направление, отмерив 116,57° против часовой стрелки от положительного направления оси x и проведя линию в этом направлении (рис. 2.25). Наш вектор (3, 116,57°) лежит где-то на этой прямой.

Следующий шаг — линейкой отмерить на этой прямой расстояние в три единицы от начала координат и поставить точку. После этого, как показано на рис. 2.26, можно измерить компоненты и получить приблизительные координаты (–1,34, 2,68).

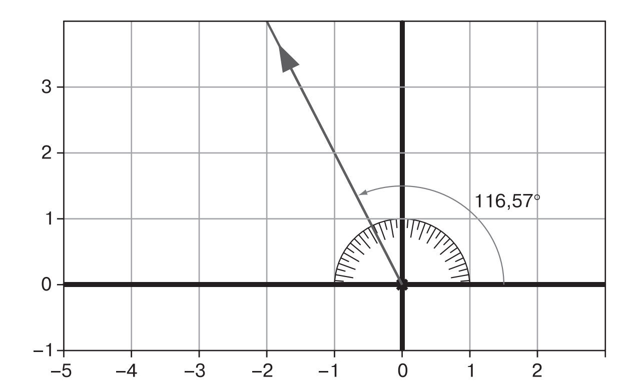


Рис. 2.25. Откладывание угла 116,57° от положительного направления оси *x* с помощью транспортира

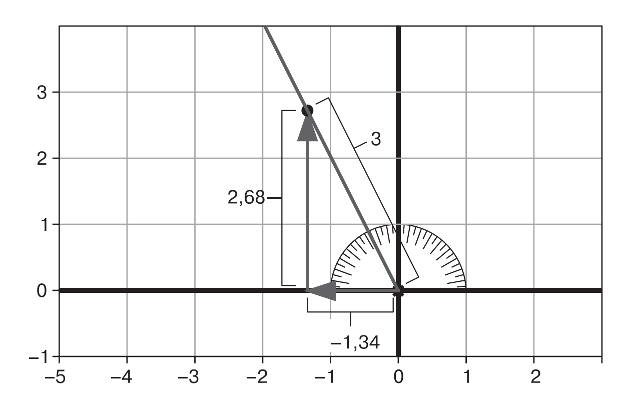


Рис. 2.26. Измерение линейкой координат точки, находящейся в трех единицах от начала координат

Может показаться, что угол 116,57° был выбран случайно, но в действительности он обладает очень интересным свойством. Двигаясь в этом направлении, вы поднимаетесь на две единицы каждый раз, когда смещаетесь на одну единицу влево. На этой линии, например, лежат векторы (–1, 2), (–3, 6) и, конечно же, (–1,34, 2,68), у каждого из них координата y в два раза больше абсолютного значения координаты x (рис. 2.27).

Странный угол 116,57° дает нам хороший коэффициент округления –2. Далеко не каждый угол дает подобные целочисленные отношения, но каждый угол дает определенное *постоянное* отношение. Угол 45° дает смещение на одну единицу по вертикали при смещении на одну единицу по горизонтали, или коэффициент 1. На рис. 2.28 показан угол 200°. Он дает смещение на –0,36 единицы по вертикали на каждую –1 единицу по горизонтали, или отношение 0,36.

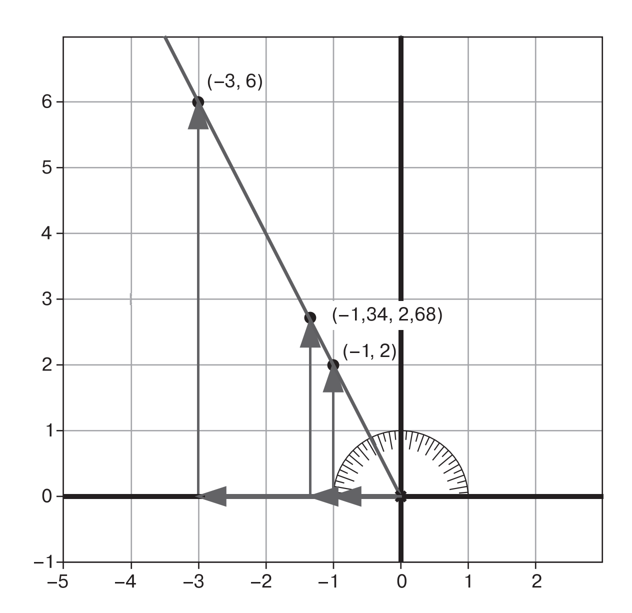


Рис. 2.27. Двигаясь в направлении 116,57°, вы поднимаетесь на две единицы при смещении на одну единицу влево

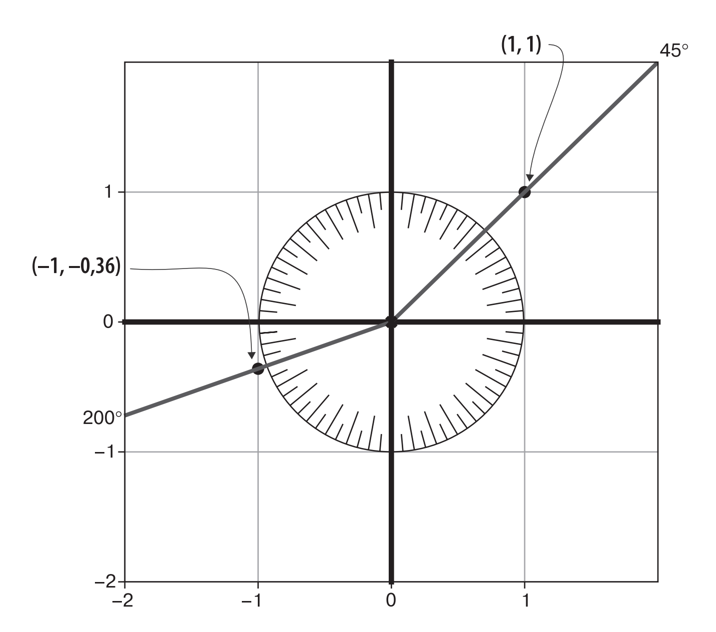


Рис. 2.28. Смещение по вертикали, которое дают разные углы при смещении на одну единицу по горизонтали

Все векторы, имеющие один и тот же угол с осью x, будут иметь постоянное отношение между координатами x и y. Это отношение называется *тангенсом* угла,

а функция тангенса обозначается tan. Вы уже видели несколько приблизительных значений этой функции:

$$\tan 37° \approx 3/4;$$
$$\tan 116{,}57° \approx -2;$$
$$\tan 45° = 1;$$
$$\tan 200° \approx 0{,}36.$$

Здесь для обозначения приблизительного равенства я использую символ «≈» вместо «=». Функция тангенса — это *тригонометрическая* функция, потому что помогает измерять треугольники. («Тригон» в слове «тригонометрия» означает треугольник, а «метрия» — измерение.) Обратите внимание: я еще не сказал вам, *как* вычислить тангенс, а только показал некоторые из его значений. В Python есть встроенная функция тангенса, которую я вскоре рассмотрю, поэтому вам почти никогда не придется беспокоиться о вычислении (или измерении) тангенса угла вручную.

Функция тангенс напрямую связана с нашей первоначальной задачей определения декартовых координат вектора по углу и расстоянию. Но она дает не координаты непосредственно, а только их отношение. Для определения фактических координат нам пригодятся еще две тригонометрические функции — *синус* и *косинус*. Если измерить некоторое расстояние под некоторым углом (рис. 2.29), то тангенс угла даст отношение расстояния, пройденного по вертикали, к расстоянию, пройденному по горизонтали.

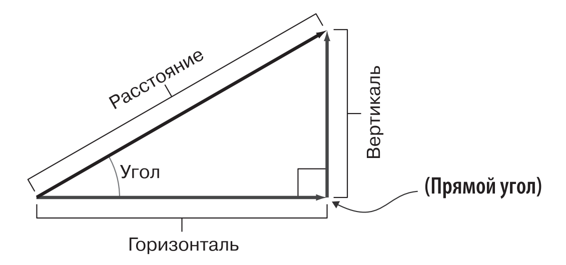


Рис. 2.29. Схема расстояний и углов для данного вектора

Для сравнения: синус и косинус дают пройденные расстояния по вертикали и горизонтали относительно общего расстояния. Они обозначаются sin и cos, и следующее уравнение демонстрирует их определения:

$$\sin(\textbf{угол}) = \frac{\textbf{вертикаль}}{\textbf{расстояние}};\ \cos(\textbf{угол}) = \frac{\textbf{горизонталь}}{\textbf{расстояние}}.$$

Рассмотрим конкретный пример — угол 37° (рис. 2.30). Мы видели, что точка (4, 3) лежит на расстоянии 5 единиц от начала координат на прямой, образующей этот угол с положительным направлением оси x.

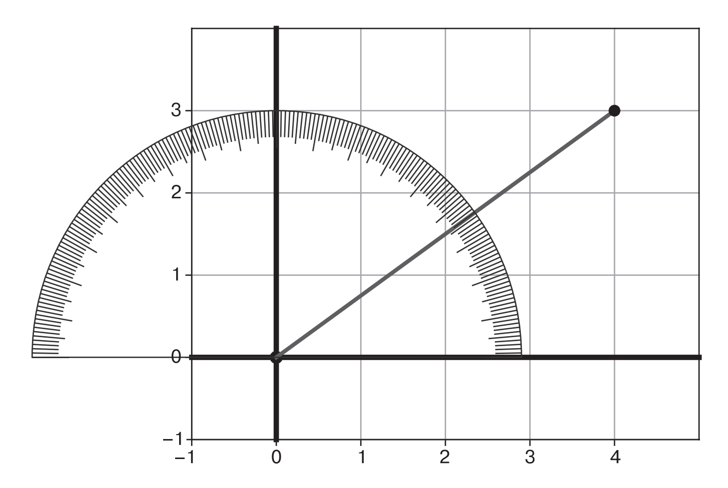


Рис. 2.30. Измерение угла для линии, соединяющей точку (4, 3) и начало координат, с помощью транспортира

На каждые 5 единиц вдоль линии, проведенной под углом 37°, приходятся примерно 3 единицы по вертикали, соответственно, можно написать:

$$\sin 37^\circ \approx 3/5.$$

Аналогично, на каждые 5 единиц вдоль линии, проведенной под углом 37°, приходятся примерно 4 единицы по горизонтали, соответственно, можно написать:

$$\cos 37^\circ \approx 4/5.$$

Это универсальная стратегия преобразования полярных координат вектора в декартовы. Если известны синус и косинус угла θ (греческая буква «тета», обычно используемая для обозначения углов) и расстояние r в этом направлении, то декартовы координаты вычисляются по формулам $r \cos \theta$ и $r \sin \theta$, как показано на рис. 2.31.

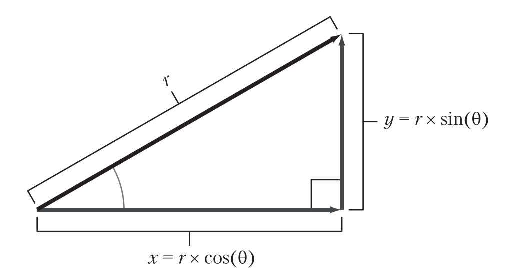


Рис. 2.31. Преобразование полярных координат в декартовы для прямоугольного треугольника

2.3.2. Радианы и тригонометрия в Python

Превратим все, что узнали о тригонометрии, в код на Python. В частности, создадим функцию, которая принимает полярные координаты (длину и угол) и возвращает декартовы координаты (длины компонент x и y).

Основное препятствие заключается в том, что встроенные тригонометрические функции в Python используют единицы измерения, отличные от тех, что применяли мы. Например, мы ожидаем, что $\tan 45° = 1$, но Python дает совсем другой результат:

```
>>> from math import tan
>>> tan(45)
1.6197751905438615
```

Python, как и большинство математиков, для измерения углов использует не градусы, а единицы, называемые *радианами*. Вот примерное соотношение между этими двумя единицами:

$$1 \text{ радиан} \approx 57{,}296°.$$

Кому-то это отношение может показаться произвольным. Однако это не так. Отношение между градусами и радианами приобретает четкий смысл, если рассматривать его в терминах специального числа π, значение которого приблизительно равно 3,14159. Вот несколько примеров:

$$\pi \text{ рад} = 180°;$$

$$2\pi \text{ рад} = 360°.$$

Половина оборота окружности составляет угол π рад, а полный оборот — 2π рад. Они соответствуют также половине и всей длине окружности с радиусом 1 (рис. 2.32).

Радианы можно представить немного иначе: для заданного угла величина в радианах сообщает, сколько радиусов вы бы прошли вдоль линии окружности. Благодаря такой независимости от единиц углы измеряются в радианах. Учитывая, что $45° = \pi/4$ рад, можно вычислить правильный результат для тангенса этого угла:

```
>>> from math import tan, pi
>>> tan(pi/4)
0.9999999999999999
```

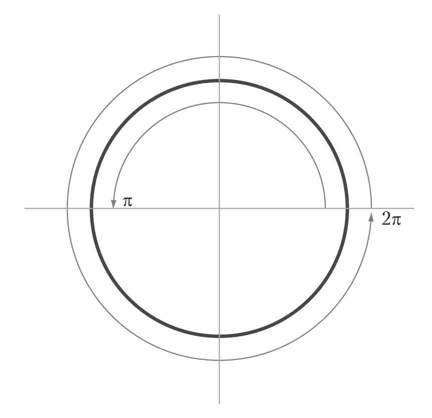


Рис. 2.32. Половина оборота окружности составляет угол π рад, а полный оборот — 2π рад

Теперь мы можем использовать тригонометрические функции Python и написать функцию `to_cartesian`, принимающую полярные координаты и возвращающую декартовы:

```
from math import sin, cos
def to_cartesian(polar_vector):
    length, angle = polar_vector[0], polar_vector[1]
    return (length*cos(angle), length*sin(angle))
```

Проверим ее, передав длину вектора 5 и угол 37°. В ответ функция должна вернуть точку (4, 3):

```
>>> from math import pi
>>> angle = 37*pi/180
>>> to_cartesian((5,angle))
(3.993177550236464, 3.0090751157602416)
```

Теперь, имея функцию преобразования полярных координат в декартовы, рассмотрим преобразование в обратном направлении.

2.3.3. От компонентов к углам

Имея декартовы координаты, такие как (–2, 3), можно найти длину вектора, применив теорему Пифагора. Это первая из двух искомых полярных координат. Вторая координата — это угол, который можно назвать θ, указывающий направление вектора (рис. 2.33).

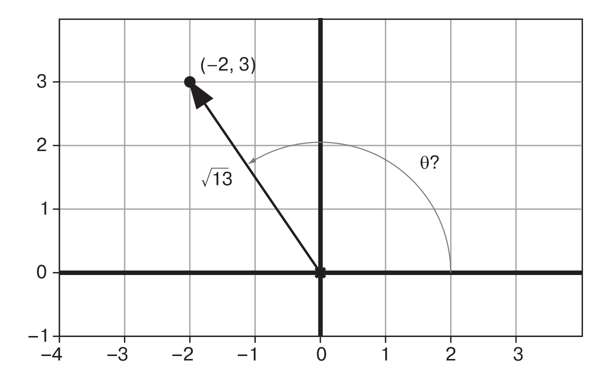


Рис. 2.33. Как определить угол, образованный вектором (–2, 3)?

Мы можем отметить некоторые данные об искомом угле θ. Его tan θ равен 3/2, $\sin\theta = 3/\sqrt{13}$ и $\cos\theta = -2/\sqrt{13}$. Осталось только найти величину θ, соответствующую этим значениям. Если хотите, то можете сделать паузу и попробовать вычислить приблизительную величину этого угла самостоятельно.

В идеале нужен более эффективный метод, чем этот. Было бы здорово иметь функцию, которая, например, принимает значение sin θ и возвращает θ. Как

оказывается, все не так просто, тем не менее в Python есть функция `math.asin`, которая делает нечто подобное. Это реализация *обратной тригонометрической функции*, называемой *арксинусом*. Она возвращает искомое значение θ:

```
>>> from math import asin
>>> sin(1)
0.8414709848078965
>>> asin(0.8414709848078965)
1.0
```

Выглядит неплохо. А как она справится с нашим синусом угла $3 / \sqrt{13}$?

```
>>> from math import sqrt
>>> asin(3/sqrt(13))
0.9827937232473292
```

Этот угол примерно равен 56,3° и, как показано на рис. 2.34, имеет совершенно неверное направление!

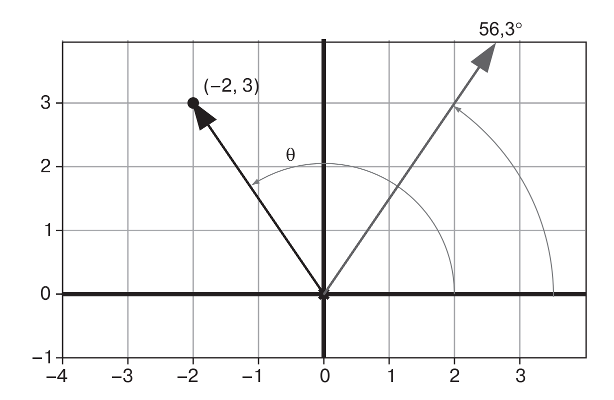


Рис. 2.34. Функция math.asin в Python, похоже, дает неверный результат

Однако это не говорит о том, что `math.asin` дает неверный результат, — *в действительности* на линии с этим направлением лежит другая точка (2, 3). Этот угол находится на расстоянии $\sqrt{13}$ от начала координат, поэтому его синус тоже равен $3 / \sqrt{13}$. Именно по этой причине `math.asin` не становится окончательным решением. Разные углы могут иметь один и тот же синус.

Правильное значение дает обратная тригонометрическая функция, называемая *арккосинусом* и реализованная в Python как `math.acos`:

```
>>> from math import acos
>>> acos(-2/sqrt(13))
2.1587989303424644
```

Это число радиан примерно равно 123,7°, что можно подтвердить с помощью транспортира. Но это случайность, есть и другие векторы, образующие углы с тем же косинусом. Например, точка (–2, –3) тоже находится на расстоянии $\sqrt{13}$

от начала координат, и соответствующий вектор образует угол с тем же косинусом, что и θ: $-2 / \sqrt{13}$. Чтобы найти верное значение θ, нужно убедиться, что синус и косинус согласуются с тем, чего мы ожидаем. Возвращаемый Python угол, который составляет примерно 2,159, удовлетворяет этому:

```
>> cos(2.1587989303424644)
-0.5547001962252293
>>> -2/sqrt(13)
-0.5547001962252291
>>> sin(2.1587989303424644)
0.8320502943378435
>>> 3/sqrt(13)
0.8320502943378437
```

Ни одной из функций — арксинуса, арккосинуса или арктангенса — недостаточно, чтобы найти угол для точки на плоскости. Можно найти правильный угол с помощью хитрого геометрического доказательства, которое вы, вероятно, учили на уроках геометрии в школе. Можете попробовать вспомнить его самостоятельно, а мы перейдем к делу и используем Python, чтобы он выполнил всю работу за нас! Функция `math.atan2` принимает декартовы координаты точки на плоскости (в обратном порядке!) и возвращает угол, который образует соответствующий вектор, например:

```
>>> from math import atan2
>>> atan2(3,-2)
2.158798930342464
```

Я прошу прощения за то, что тянул кота за хвост, но вы должны знать о потенциальных ловушках при использовании обратных тригонометрических функций: они могут возвращать один и тот же результат для разных входных данных, поэтому по результату нельзя точно определить, какими были входные данные. Новые знания позволяют нам завершить функцию преобразования декартовых координат в полярные, которую мы намеревались написать:

```
def to_polar(vector):
    x, y = vector[0], vector[1]
    angle = atan2(y,x)
    return (length(vector), angle)
```

Проверим ее на нескольких простых примерах: вызов `to_polar((1,0))` должен вернуть одну единицу в положительном направлении x и угол 0°. И действительно, вызов функции возвращает длину, равную единице, и угол, равный нулю:

```
>>> to_polar((1,0))
(1.0, 0.0)
```

(Совпадение входных и выходных данных — случайность: они имеют разный геометрический смысл.) Аналогично получаем ожидаемый ответ для (–2, 3):

```
>>> to_polar((-2,3))
(3.605551275463989, 2.158798930342464)
```

2.3.4. Упражнения

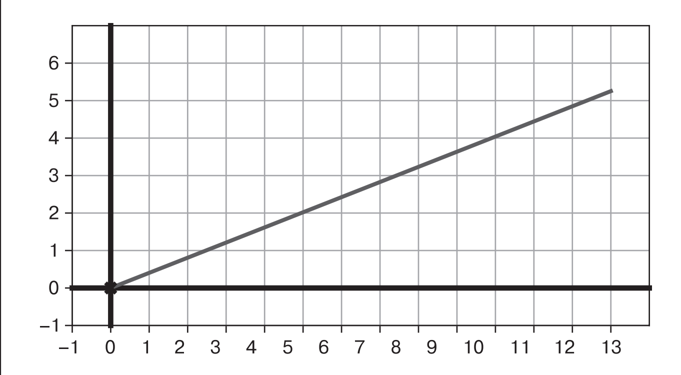
Упражнение 2.27. Убедитесь, что вектор, заданный декартовыми координатами (–1,34, 2,68), имеет длину, примерно равную 3.

Решение.

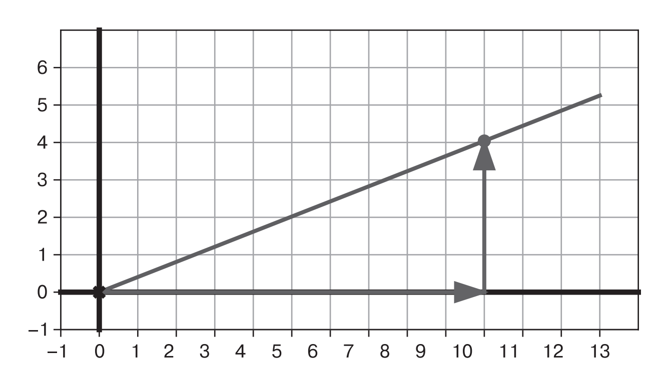
```
>>> length((-1.34,2.68))
2.9963310898497184
```

Очень близко!

Упражнение 2.28. На рисунке показана линия, образующая угол 22° с положительным направлением оси x. Опираясь на этот рисунок, определите примерное значение tan 22°.



Решение. Линия проходит близко к точке (10, 4), как показано далее, поэтому значение tan 22° примерно равно 4/10 = 0,4.



Упражнение 2.29. Попробуйте ответить на обратный вопрос. Пусть известны длина и направление вектора и требуется найти его компоненты. Определите компоненты x и y вектора, который имеет длину 15 и образует угол 37°.

Решение.

sin 37° равен примерно 3/5, то есть на каждые 5 единиц длины вектора он поднимается над осью x на 3 единицы. Так что при длине 15 единиц мы получаем вертикальную компоненту $3/5 \cdot 15 = 9$.

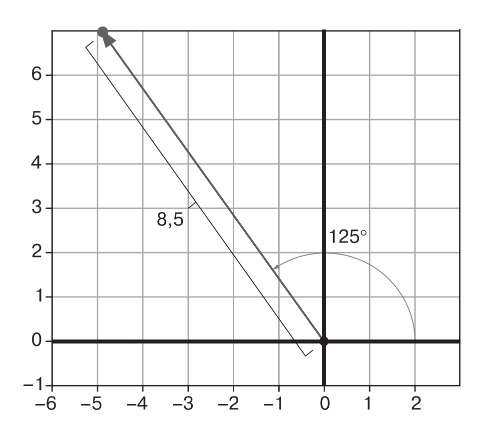
cos 37° примерно равен 4/5, то есть на каждые 5 единиц длины вектор смещается на 4 единицы вправо, поэтому горизонтальный компонент равен $4/5 \cdot 15$, или 12. Таким образом, полярные координаты (15, 37°) примерно соответствуют декартовым координатам (12, 9).

Упражнение 2.30. Пусть имеется вектор длиной 8,5 единицы, образующий угол 125° с положительным направлением оси x. Учитывая, что sin 125° = 0,819 и cos 125° = –0,574, определите декартовы координаты вектора. Покажите на рисунке вектор и образованный им угол.

Решение.

$$x = r\cos\theta = 8{,}5 \cdot (-0{,}574) = -4{,}879;$$
$$y = r\sin\theta = 8{,}5 \cdot 0{,}819 = 6{,}962.$$

На рисунке показана точка с координатами (–4,879, 6,962).



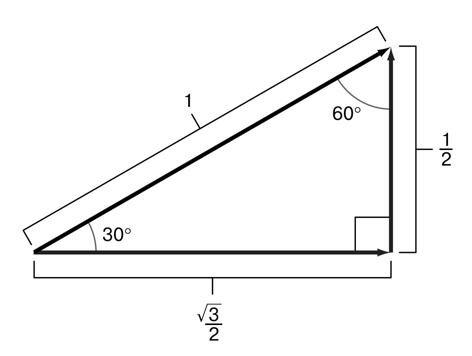
Упражнение 2.31. Чему равны синус и косинус 0°? 90°? 180°? Иначе говоря, сколько единиц по вертикали и горизонтали приходится на единицу длины вектора, образующего любой из этих углов?

Решение. Вектор, образующий угол 0° с осью x, лежит на самой оси x, то есть его вертикальная компонента равна 0, соответственно $\sin 0° = 0$. А так как на каждую единицу длины приходится ровно одна единица смещения по горизонтали вправо, то $\cos 0° = 1$.

Вектор, образующий угол 90° (четверть оборота против часовой стрелки), лежит на оси y, соответственно, на каждую единицу длины приходится ровно одна единица смещения по вертикали вверх, поэтому $\sin 90° = 1$, а $\cos 90° = 0$.

Наконец, вектор, образующий угол 180°, лежит на оси x и направлен в сторону отрицательных значений, поэтому на каждую единицу длины приходится ровно одна единица смещения по горизонтали влево, соответственно, $\cos 180° = -1$ и $\sin 180° = 0$.

Упражнение 2.32. На следующей схеме показаны некоторые точные размеры прямоугольного треугольника.



Во-первых, подтвердите, что эти размеры действительны для прямоугольного треугольника, то есть удовлетворяют теореме Пифагора. Затем вычислите значения $\sin 30°$, $\cos 30°$ и $\tan 30°$ с точностью до трех знаков после запятой, используя размеры, указанные на схеме.

Решение. Эти длины сторон действительно удовлетворяют теореме Пифагора:

$$\sqrt{\left(\frac{1}{2}\right)^2+\left(\frac{\sqrt{3}}{2}\right)^2}=\sqrt{\frac{1}{4}+\frac{3}{4}}=\sqrt{\frac{4}{4}}=1.$$

Подстановка длин сторон в теорему Пифагора

Значения тригонометрических функций определяются отношениями соответствующих длин сторон:

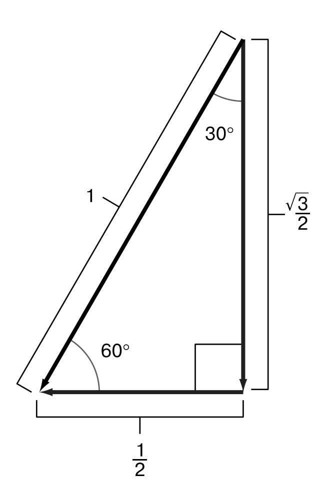
$$\sin 30° = \left(\frac{1}{2}\right)\Big/1 = 0{,}5;$$

$$\cos 30° = \left(\frac{\sqrt{3}}{2}\right)\Big/1 \approx 0{,}866;$$

$$\tan 30° = \left(\frac{1}{2}\right)\Big/\left(\frac{\sqrt{3}}{2}\right) \approx 0{,}577.$$

Вычисление синуса, косинуса и тангенса согласно их определениям

Упражнение 2.33. Взгляните на треугольник из предыдущего упражнения с другой точки зрения и вычислите значения sin 60°, cos 60° и tan 60° с точностью до трех знаков после запятой.



Повернутая копия треугольника из предыдущего упражнения

Решение. Поворот и отражение треугольника из предыдущего упражнения не влияют ни на длины его сторон, ни на углы.

Отношения длин сторон дают следующие значения тригонометрических функций для угла 60°:

$$\sin 60° = \left(\frac{\sqrt{3}}{2}\right) \Big/ 1 \approx 0{,}866;$$

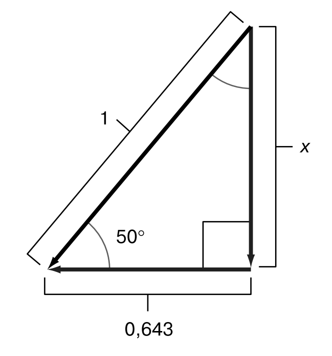
$$\cos 60° = \left(\frac{1}{2}\right) \Big/ 1 = 0{,}5;$$

$$\tan 60° = \left(\frac{\sqrt{3}}{2}\right) \Big/ \left(\frac{1}{2}\right) \approx 1{,}732.$$

Вычисление отношений после перемены местами горизонтальной и вертикальной компонент

Упражнение 2.34. Косинус угла 50° равен 0,643. Чему равен sin 50° и tan 50°? Нарисуйте схему, чтобы упростить получение ответа.

Решение. С учетом того, что cos 50° = 0,643, соответствующий прямоугольный треугольник будет выглядеть так.



То есть отношение длин двух известных сторон будет 0,643/1 = 0,643. Чтобы найти неизвестную длину третьей стороны, можно воспользоваться теоремой Пифагора:

$$\sqrt{0{,}643^2 + x^2} = 1;$$

$$0{,}643^2 + x^2 = 1;$$

$$0{,}413 + x^2 = 1;$$

$$x^2 = 0{,}587;$$

$$x = 0{,}766.$$

Определив длины всех сторон, получаем: sin 50° = 0,766/1 = 0,766, tan 50° = 0,766/0,643 = 1,192.

Упражнение 2.35. Выразите угол 116,57° в радианах. Используйте Python, чтобы вычислить тангенс этого угла, и убедитесь, что он близок к –2, как было показано ранее.

Решение. 116,57° · (1 рад/57,296°) = 2,035 рад:

```
>>> from math import tan
>>> tan(2.035)
-1.9972227673316139
```

Упражнение 2.36. Найдите угол $10\pi/6$. Какими, по вашему мнению, получатся значения $\cos(10\pi/6)$ и $\sin(10\pi/6)$ — положительными или отрицательными? Используйте Python для вычисления этих значений и подтвердите или опровергните свои догадки.

Решение. Полный оборот равен 2π рад, поэтому угол $\pi/6$ равен 1/12 окружности. Представьте разрезание пиццы на 12 частей и считайте от положительного направления оси *x* против часовой стрелки, угол $10\pi/6$ — на два реза меньше полного оборота. То есть соответствующий рез направлен вниз и вправо. Косинус должен быть положительным, а синус — отрицательным, потому что движение вдоль этого реза соответствует положительному смещению по горизонтали и отрицательному по вертикали:

```
>>> from math import pi, cos, sin
>>> sin(10*pi/6)
-0.8660254037844386
>>> cos(10*pi/6)
0.5000000000000001
```

Упражнение 2.37. Следующий генератор списков создает полярные координаты 1000 точек:

```
[(cos(5*x*pi/500.0), 2*pi*x/1000.0) for x in range(0,1000)]
```

Напишите код на Python, преобразующий их в декартовы координаты, нарисуйте получившиеся точки и соедините их замкнутой ломаной линией, чтобы получить изображение.

Решение. Вот код, включающий предварительную подготовку исходных данных:

```
polar_coords = [(cos(x*pi/100.0), 2*pi*x/1000.0) for x in range(0,1000)]
vectors = [to_cartesian(p) for p in polar_coords]
draw(Polygon(*vectors, color=green))
```

А так выглядит рисунок, который должен получиться, — пятилепестковый цветок.

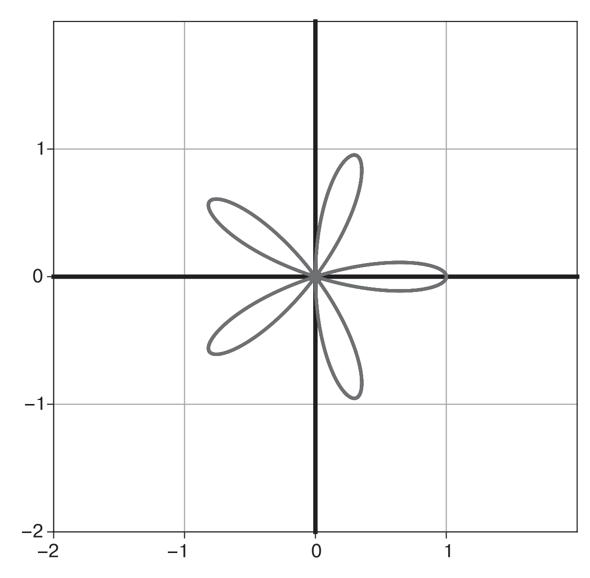
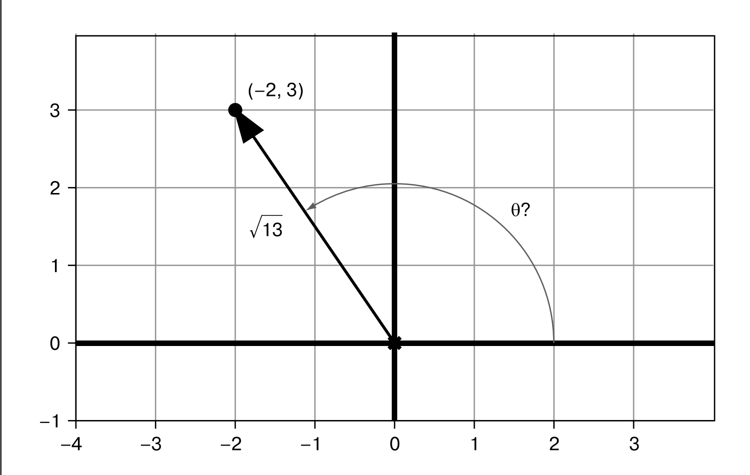


Рисунок в форме цветка, полученный соединением отрезками 1000 точек

Упражнение 2.38. Найдите угол, который образует вектор (−2, 3) с положительным направлением оси x, методом подбора.



Какой угол образует вектор (−2, 3)?

Подсказка. Визуально видно, что ответ находится между $\pi/2$ и π. В этом интервале значения синуса и косинуса всегда уменьшаются с увеличением угла.

Решение. Вот пример подбора значения между $\pi/2$ и π в поисках угла с тангенсом, близким к $-3/2 = -1{,}5$:

```
>>> from math import tan, pi
>>> pi, pi/2
(3.141592653589793, 1.5707963267948966)
>>> tan(1.8)
-4.286261674628062
>>> tan(2.5)
-0.7470222972386603
>>> tan(2.2)
-1.3738230567687946
>>> tan(2.1)
-1.7098465429045073
>>> tan(2.15)
-1.5289797578045665
>>> tan(2.16)
-1.496103541616277
>>> tan(2.155)
-1.5124173422757465
>>> tan(2.156)
-1.5091348993879299
>>> tan(2.157)
-1.5058623488727219
>>> tan(2.158)
-1.5025996395625054
>>> tan(2.159)
-1.4993467206361923
```

Значение должно быть между 2,158 и 2,159.

Упражнение 2.39. Найдите другой вектор на плоскости, образующий с осью x угол, тангенс которого равен θ, а именно $-3/2$. Используя реализацию функции *арктангенса* на Python, `math.atan`, найдите значение этого угла.

Решение. Другой вектор, образующий угол с тангенсом $-3/2$, — это вектор $(3, -2)$. Функция `math.atan` возвращает величину угла:

```
>>> from math import atan
>>> atan(-3/2)
-0.982793723247329
```

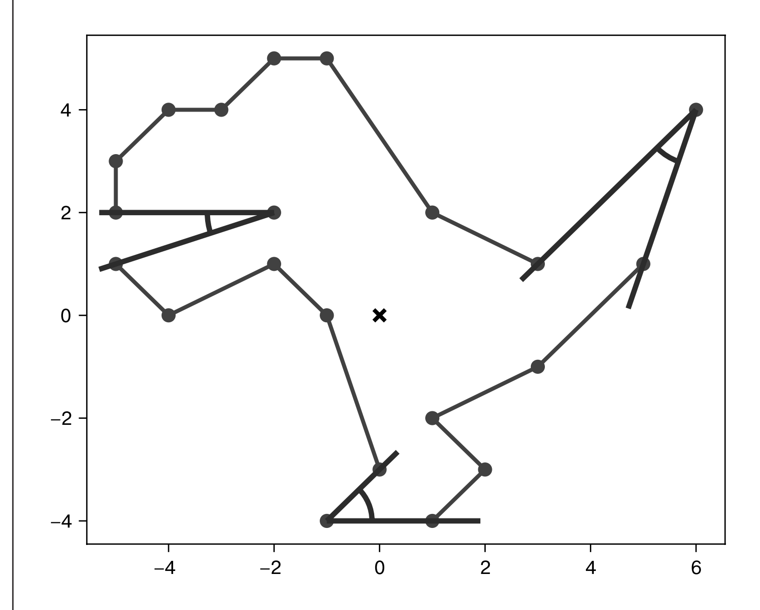
Это чуть меньше четверти оборота по часовой стрелке.

Упражнение 2.40. Не используя Python, определите, какие полярные координаты соответствуют декартовым координатам (1, 1) и (1, –1). Проверьте полученные результаты с помощью функции `to_polar`.

Решение. Полярным координатам (1, 1) соответствуют декартовы координаты $\left(\sqrt{2}, \pi/4\right)$, а полярным координатам (1, –1) — декартовы координаты $\left(\sqrt{2}, -\pi/4\right)$.

Будучи внимательным, можно найти угол между любой парой векторов, образующих фигуру. Угол между двумя векторами может определяться либо как сумма, либо как разность углов, которые они образуют с осью x. В следующем мини-проекте вам будет предложено измерить более сложные углы.

Упражнение 2.41. Мини-проект. Определите угол между отрезками, образующими пасть динозавра, палец на ноге, кончик хвоста.



Некоторые углы на изображении динозавра, которые можно измерить или вычислить

2.4. ПРЕОБРАЗОВАНИЕ НАБОРОВ ВЕКТОРОВ

Наборы векторов хранят пространственные данные, такие как рисунки динозавров, не зависящие от используемой системы координат — полярной или декартовой. Когда дело доходит до манипулирования векторами, одна система координат может оказаться удобнее другой. Мы уже видели, что параллельный перенос набора векторов проще выполнять с декартовыми координатами. В полярных координатах вычисления выглядят намного менее естественными. Однако с такими координатами проще выполнять повороты, потому что они имеют встроенные углы.

Прибавление константы к значению угла в полярных координатах поворачивает вектор против часовой стрелки, а вычитание — по часовой стрелке. Полярные координаты (1, 2) описывают точку, находящуюся на расстоянии 1 и под углом 2 рад. (Помните, что мы работаем с радианами, если в записи отсутствует символ градуса!) Прибавляя к величине угла или вычитая из нее 1, мы поворачиваем вектор на 1 рад против часовой стрелки или по часовой стрелке соответственно (рис. 2.35).

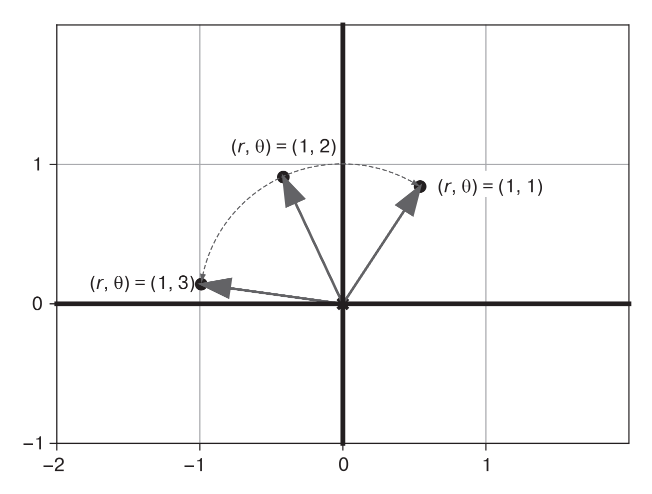


Рис. 2.35. Прибавление значения к величине угла или его вычитание из нее поворачивает вектор вокруг начала координат

Одновременный поворот нескольких векторов приводит к повороту вокруг начала координат всей фигуры, которую они представляют. Функция `draw` понимает только декартовы координаты, поэтому перед ее вызовом нужно преобразовать полярные координаты в декартовы. Точно так же, поскольку поворачивать векторы удобнее в полярных координатах, перед выполнением поворота мы должны преобразовать декартовы координаты в полярные. Используя этот подход, можно повернуть нашего динозавра:

```
rotation_angle = pi/4
dino_polar = [to_polar(v) for v in dino_vectors]
```

```
dino_rotated_polar = [(l,angle + rotation_angle) for l,angle in dino_polar]
dino_rotated = [to_cartesian(p) for p in dino_rotated_polar]
draw(
    Polygon(*dino_vectors, color=gray),
    Polygon(*dino_rotated, color=red)
)
```

Результат выполнения этого кода — светло-серая копия оригинального динозавра с наложенной темно-серой копией, повернутой на угол $\pi/4$, или на $1/8$ часть полного оборота против часовой стрелки (рис. 2.36).

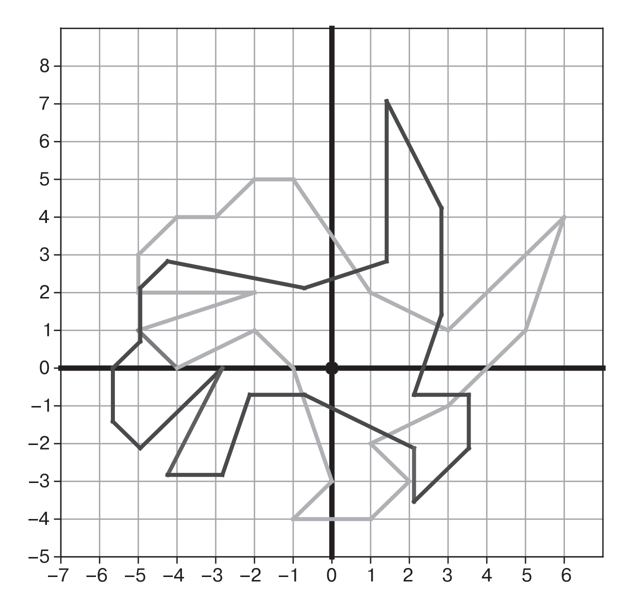


Рис. 2.36. Оригинальный динозавр и повернутая копия

В качестве упражнения, завершающего этот раздел, напишите универсальную функцию `rotate`, которая поворачивает список векторов на один и тот же угол. Я буду использовать такую функцию в следующих нескольких примерах, а вы можете применить как мою реализацию, так и написанную собственноручно.

2.4.1. Комбинирование векторных преобразований

До сих пор мы рассматривали только параллельный перенос и поворот векторов. Применение любого из этих преобразований к набору векторов оказывает такое же воздействие на всю фигуру на плоскости. Но полная мощь этих векторных преобразований проявляется, если их задействовать последовательно.

Например, можно сначала повернуть динозавра, а *затем* выполнить параллельный перенос. Такое преобразование можно кратко записать, используя функцию

`translate` из упражнения из раздела 2.2.4 и функцию `rotate` (см. результат на рис. 2.37):

```
new_dino = translate((8,8), rotate(5 * pi/3, dino_vectors))
```

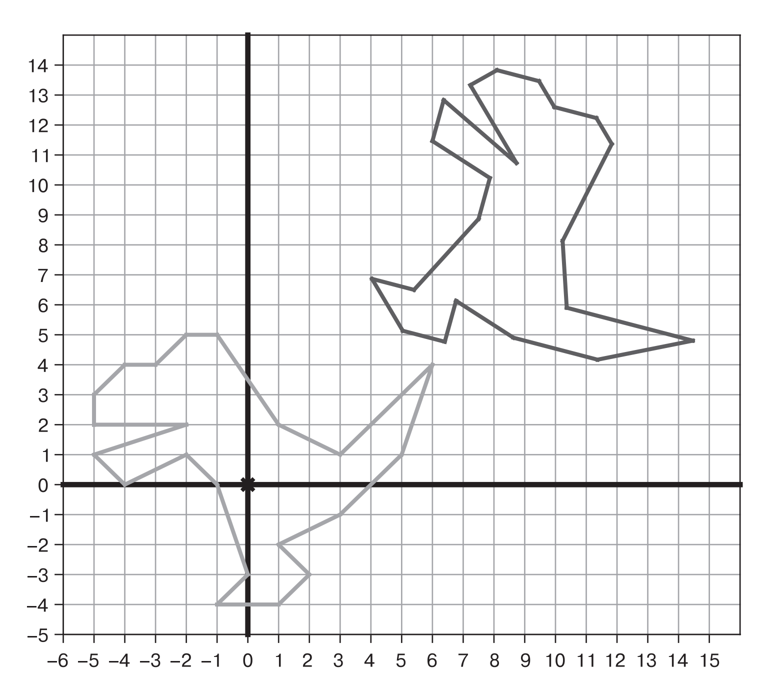


Рис. 2.37. Оригинальный динозавр и копия, сначала повернутая, а потом смещенная

Первым выполняется поворот против часовой стрелки на $5\pi/3$, то есть чуть меньше, чем на полный оборот. Затем производится параллельный перенос вверх и вправо на 8 единиц в каждом направлении. Как нетрудно догадаться, правильно сочетая повороты и переносы, можно переместить динозавра (или другую фигуру) в любое желаемое место на плоскости и придать ему (ей) нужную ориентацию. Независимо от того, где производится преобразование — в фильме или игре, — мы можем перемещать динозавра с помощью векторных преобразований и анимировать его программно.

Вскоре мы закончим эксперименты с мультяшными динозаврами — существует множество других операций с векторами, и многие из них легко обобщаются на большее число измерений. Наборы данных в реальном мире часто имеют десятки и сотни измерений, и мы будем применять к ним те же преобразования. Зачастую бывает полезно выполнить параллельный перенос и поворот набора данных, чтобы сделать их отличительные черты особенно заметными. Мы не можем изобразить поворот в 100 измерениях, но всегда можем использовать представление поворота в двух измерениях как надежную метафору.

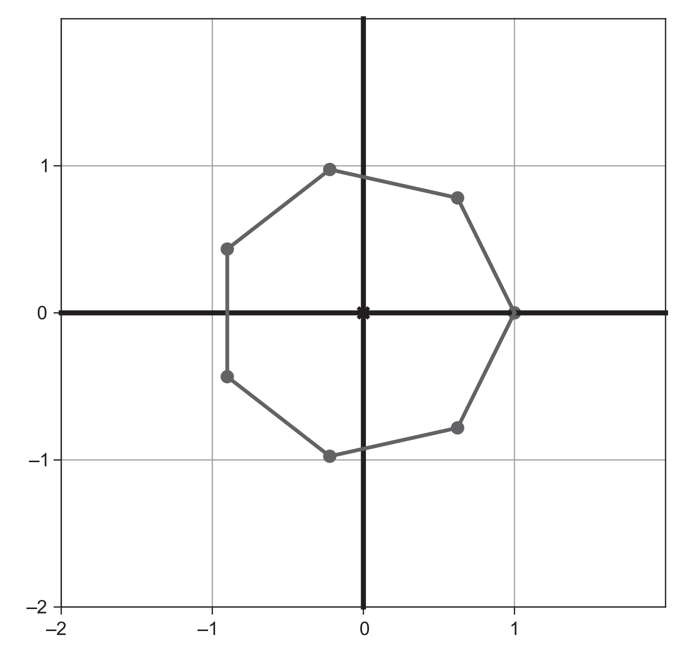
2.4.2. Упражнения

Упражнение 2.42. Напишите функцию `rotate(angle, vectors)`, принимающую массив `vectors` декартовых координат векторов и поворачивающую их на угол `angle` (против часовой стрелки или по часовой стрелке в зависимости от знака угла).

Решение

```
def rotate(angle, vectors):
    polars = [to_polar(v) for v in vectors]
    return [to_cartesian((l, a+angle)) for l,a in polars]
```

Упражнение 2.43. Напишите функцию `regular_polygon(n)`, которая возвращает декартовы координаты вершин правильного n-стороннего многоугольника (то есть все углы и длины сторон которого равны). Например, `polygon(7)` должна вернуть массив векторов, определяющий следующий семиугольник.



Правильный семиугольник с точками в семи углах, равноудаленных от начала координат

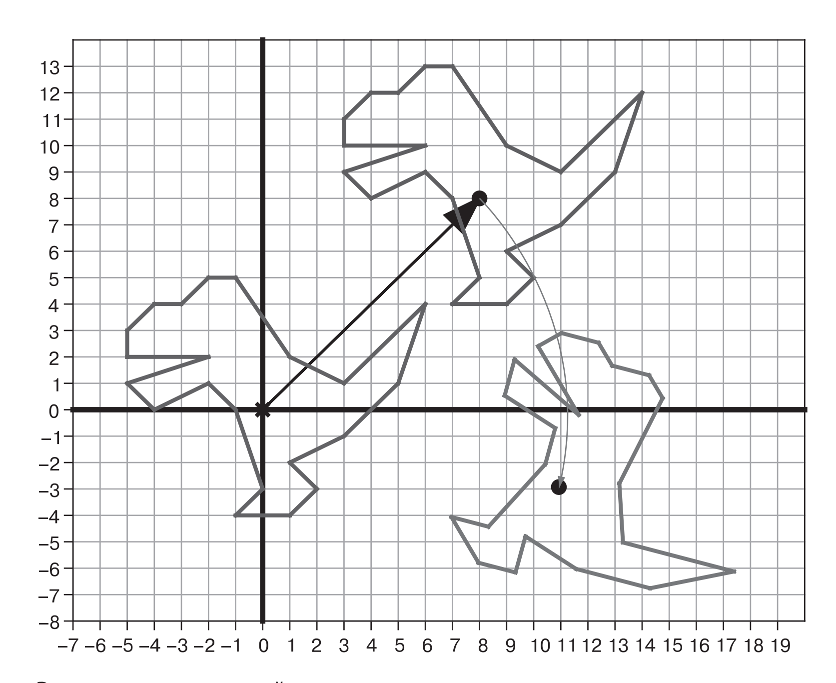
Подсказка. На этом рисунке я использовал вектор $(1, 0)$ и скопировал его семь раз, поворачивая каждую копию вокруг начала координат на $1/7$ полного оборота.

Решение

```
def regular_polygon(n):
    return [to_cartesian((1, 2*pi*k/n)) for k in range(0,n)]
```

Упражнение 2.44. Что получится, если сначала выполнить параллельный перенос динозавра вдоль вектора $(8, 8)$, а затем поворот на $5\pi/3$? Получится ли тот же результат, как в случае, когда сначала выполняется поворот, а потом перенос?

Решение



Результат, получаемый, когда сначала выполняется перенос, а потом поворот

Результат получится *не* тот же самый. В общем случае применение поворота и переноса в разном порядке дает разные результаты.

2.5. РИСОВАНИЕ С ПОМОЩЬЮ MATPLOTLIB

Как и обещал, в заключение покажу, как написать с нуля функции рисования из этой главы, которые используют библиотеку Matplotlib. После установки библиотеки Matplotlib с помощью `pip` ее (и некоторые ее подмодули) можно импортировать, например:

```
import matplotlib
from matplotlib.patches import Polygon
from matplotlib.collections import PatchCollection
```

Классы `Polygon`, `Points`, `Arrow` и `Segment` не так интересны — они просто хранят данные, полученные ими в своих конструкторах. Например, класс `Points` имеет только конструктор, который принимает и сохраняет список векторов, а также именованный аргумент `color`:

```
class Points():
    def __init__(self, *vectors, color=black):
        self.vectors = list(vectors)
        self.color = color
```

Функция `draw` начинается с определения размеров диаграммы и затем рисует объекты из списка по одному. Например, для рисования точек на плоскости, представленных объектом `Points`, используются функции рисования графиков из библиотеки Matplotlib:

```
def draw(*objects, ...
    # ...
    for object in objects:
    # ...
        elif type(object) == Points:
            xs = [v[0] for v in object.vectors]
            ys = [v[1] for v in object.vectors]
            plt.scatter(xs, ys, color=object.color)

        # ...
```

Некоторые предварительные настройки (здесь не показаны)

Обход объектов в списке

Если текущий объект — это экземпляр класса Points, то выполняется рисование всех точек в списке с помощью функции scatter bo Matplotlib

Стрелки, отрезки и многоугольники обрабатываются почти так же — с использованием различных функций из Matplotlib. Все это можно увидеть в файле с исходным кодом `vector_drawing.py`. Для построения графиков и математических функций в этой книге мы будем применять Matplotlib и постепенно расширять ее возможности.

Теперь, когда вы освоили два измерения, можно добавить еще одно. Получив третье измерение, мы сможем описывать окружающий мир, в котором живем. В следующей главе вы познакомитесь с приемами моделирования трехмерных объектов.

КРАТКИЕ ИТОГИ ГЛАВЫ

- Векторы — это математические объекты, существующие в многомерных пространствах. Это могут быть геометрические пространства, такие как двухмерная плоскость экрана компьютера или трехмерный мир, в котором мы живем.
- Векторы можно представлять как стрелки с заданными длиной и направлением или как точки на плоскости, имеющие определенные координаты относительно контрольной точки, называемой *началом координат*. Каждой точке соответствует стрелка, показывающая, как добраться до нее из начала координат.
- Точки на плоскости можно соединять отрезками и формировать интересные геометрические фигуры, например динозавра.
- Координаты на плоскости — это пары чисел, позволяющие определить местоположение точек на плоскости. Значения x и y, записанные в виде кортежа (x, y), говорят нам, как далеко по горизонтали и вертикали находится точка.
- В программном коде на Python точки можно хранить в виде кортежей и использовать различные библиотеки для их рисования на экране.
- Сложение векторов вызывает эффект параллельного переноса, или перемещения, первого вектора в направлении, в котором указывает второй вектор-слагаемое. Если набор векторов представить как описание траектории движения, то их сумма даст направление и расстояние до конечной точки.
- Скалярное умножение вектора на числовую константу дает вектор, указывающий в прежнем направлении, но длина которого равна произведению длины исходного вектора на константу.
- Вычитание векторов дает относительное положение вектора-вычитаемого относительно вектора-уменьшаемого.
- Векторы можно задавать длиной и направлением в виде угла. Эти два числа определяют полярные координаты двухмерного вектора.
- Для преобразования обычных (декартовых) координат в полярные используются тригонометрические функции синуса, косинуса и тангенса.
- Фигуры проще поворачивать, если они определяются наборами векторов с полярными координатами. Для этого нужно лишь прибавить или вычесть заданный угол поворота из величины угла каждого вектора. Поворот и перенос фигур на плоскости позволяет произвольно перемещать их по плоскости и менять ориентацию.

3

Выход в трехмерный мир

В этой главе

- ✓ Формирование мысленной модели трехмерных векторов.
- ✓ Арифметика трехмерных векторов.
- ✓ Использование скалярного и векторного произведений для определения длин и направлений.
- ✓ Отображение трехмерного объекта на двухмерной плоскости.

Двухмерный мир легко визуализировать, но реальный мир имеет три измерения. Для чего бы ни использовалось программное обеспечение — проектирования зданий, создания анимационных фильмов или компьютерных игр, — наши программы должны учитывать три пространственных измерения, в которых мы живем.

В двухмерном пространстве, подобном странице этой книги, мы имеем два направления, вертикальное и горизонтальное. Добавив третье измерение, можем также говорить о точках за пределами страницы или о стрелках, направленных перпендикулярно ей. Но даже когда программы моделируют трехмерный мир, они отображают результаты на двухмерных компьютерных дисплеях. Наша задача в этой главе — создать инструменты, необходимые для получения трехмерных объектов, представленных трехмерными векторами, и их преобразования в двухмерные изображения, которые можно отобразить на экране.

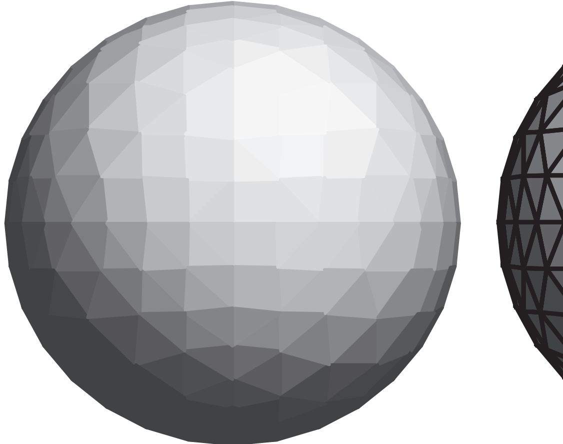
Сфера — один из примеров трехмерной формы. Удачно нарисованная трехмерная сфера может выглядеть так, как показано на рис. 3.1. Без эффекта затенения она выглядела бы как простой круг.

Затенение показывает, что свет падает на нашу сферу под определенным углом, и создает иллюзию глубины. Общая стратегия заключается не в том, чтобы нарисовать идеально круглую сферу, а в том, чтобы получить ее приближение, состоящее из многоугольников. Каждый многоугольник можно затенить с учетом угла падения лучей света на него. Хотите верьте, хотите нет, но на рис. 3.1 изображен не круглый шар, а 8000 треугольников разных оттенков. На рис. 3.2 приведен пример сферы, составленной из меньшего числа треугольников.



Рис. 3.1. Благодаря эффекту затенения двухмерный круг можно превратить в трехмерную сферу

У нас есть математический аппарат, необходимый для определения треугольника на двухмерном экране: нужны только три двухмерных вектора, определяющих углы. Но мы не сможем решить, как их заштриховать, если не будем рассматривать их как фигуры в трехмерном пространстве. А для этого нужно научиться работать с трехмерными векторами.



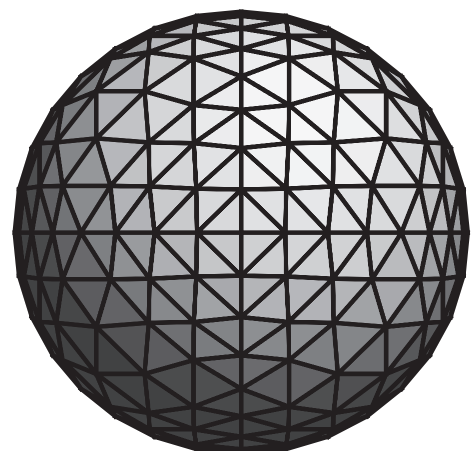


Рис. 3.2. Сфера, нарисованная с использованием меньшего числа треугольников разных оттенков

Конечно, эта задача давно решена, поэтому начнем с использования готовой библиотеки для рисования трехмерных фигур. Разобравшись в особенностях мира трехмерных векторов, мы сможем создать свои функции отображения и нарисовать сферу.

3.1. ОТОБРАЖЕНИЕ ВЕКТОРОВ В ТРЕХМЕРНОМ ПРОСТРАНСТВЕ

Работая с двухмерной плоскостью, мы задействовали три взаимозаменяемые мысленные модели векторов: пары координат, стрелки с фиксированными длиной и направлением и точки, расположенные на некоем расстоянии относительно начала координат. Поскольку страницы этой книги имеют конечный размер, мы ограничились обзором небольшой части плоскости — прямоугольником фиксированных высоты и ширины, подобным показанному на рис. 3.3.

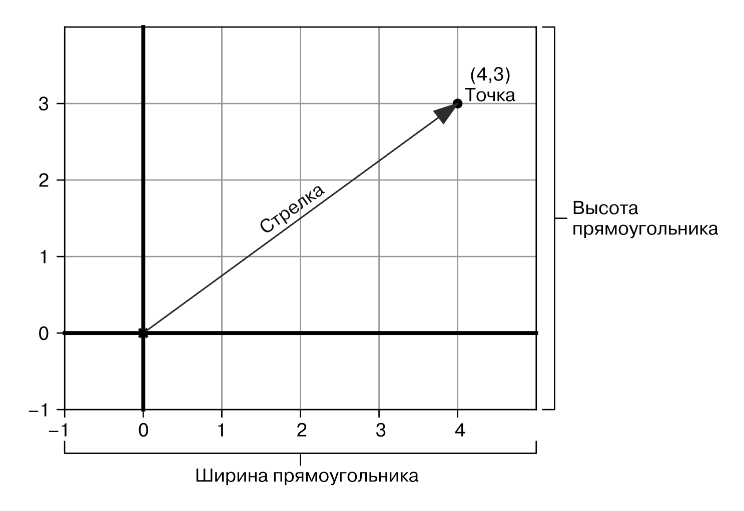


Рис. 3.3. Высота и ширина небольшой части двухмерной плоскости

Точно так же можно интерпретировать трехмерные векторы, только вместо прямоугольной части плоскости рассматривать блок трехмерного пространства. Такой трехмерный блок (рис. 3.4) имеет конечные высоту, ширину и глубину. В трехмерном мире сохраняются понятия направлений x и y и добавляется направление z — глубина.

Мы можем представить любой двухмерный вектор в трехмерном пространстве как имеющий те же размер и ориентацию, но привязанный к плоскости, то есть при глубине $z = 0$. На рис. 3.5 вверху показан двухмерный рисунок вектора (4, 3),

внедренного в трехмерное пространство со всеми его свойствами. На втором рисунке (внизу) обозначены все свойства, которые никуда не делись.

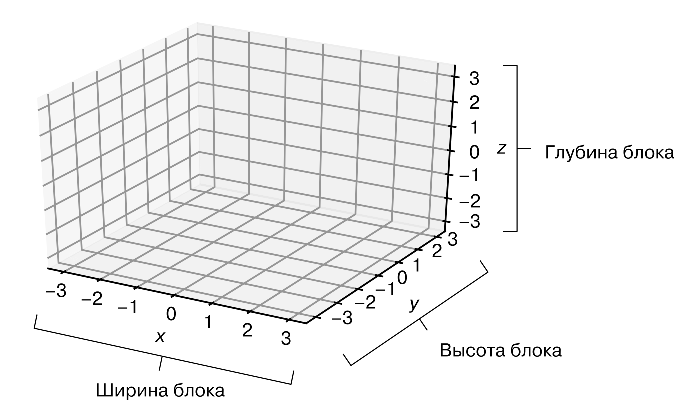
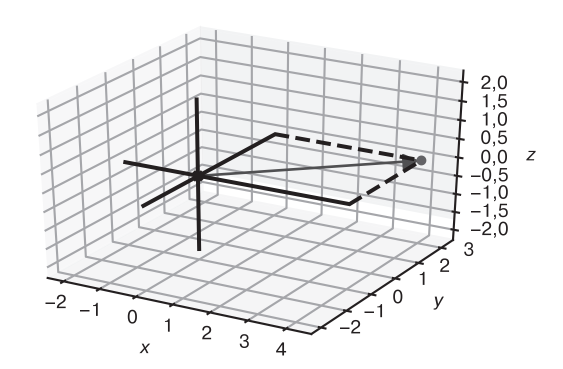


Рис. 3.4. Небольшой ограниченный блок трехмерного пространства имеет ширину (*x*), высоту (*y*) и глубину (*z*)



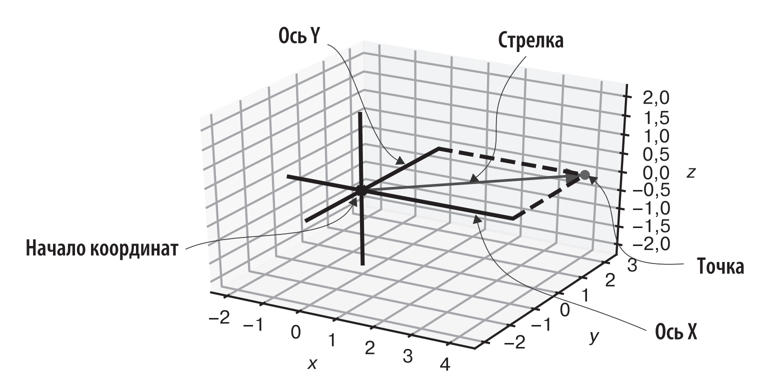


Рис. 3.5. Вектор, обитатель двухмерного мира, в трехмерном мире

Пунктирные линии образуют прямоугольник на двухмерной плоскости, где значение глубины равно нулю. Часто полезно рисовать пунктирные линии, встречающиеся под прямым углом, чтобы помочь найти точки в трехмерном пространстве. В противном случае чувство перспективы может обмануть нас и точка окажется не там, где мы думаем.

Наш вектор по-прежнему лежит на плоскости, но теперь, как видите, он находится в большем трехмерном пространстве. Мы можем нарисовать еще один вектор (новую стрелку и новую точку), выходящий за пределы исходной плоскости и имеющий ненулевое значение глубины (рис. 3.6).

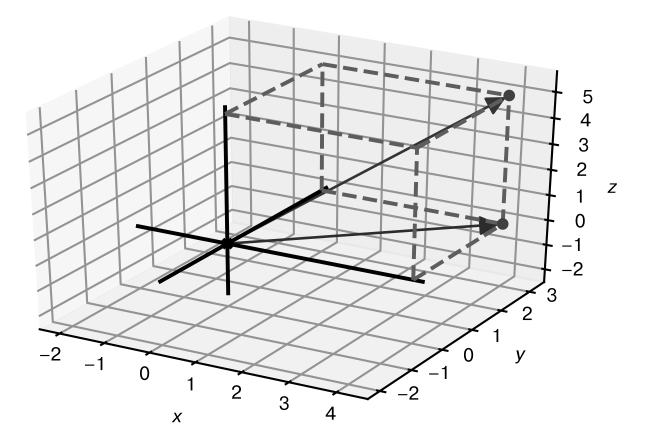


Рис. 3.6. Вектор, простирающийся в третье измерение, в сравнении с двухмерным миром и вектором (4, 3) в нем

Для большей ясности расположения этого второго вектора я обозначил пунктиром пространственный блок по аналогии с прямоугольником на рис. 3.5. На рис. 3.6 этот блок подчеркивает длину, ширину и глубину части трехмерного пространства, охватываемого вектором. Мысленные модели векторов в виде стрелок и точек работают в трехмерном пространстве точно так же, как в двухмерном, и точно так же характеризуются с помощью координат.

3.1.1. Представление трехмерных векторов с помощью координат

Чтобы указать одну точку или стрелку, в двухмерном пространстве достаточно пары чисел (4, 3), но в трехмерном мире существует множество точек с координатой x, равной 4, и координатой y, равной 3. Фактически, как показано на рис. 3.7, существует целая линия из точек с этими координатами, каждая из которых имеет разные значения глубины z.

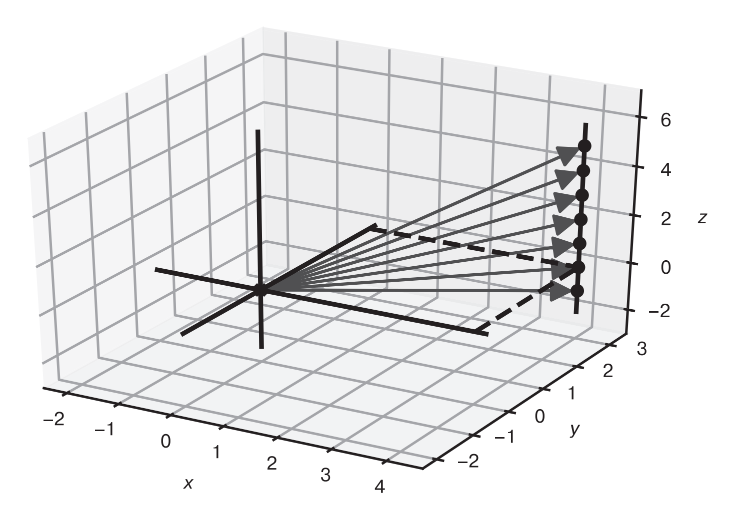


Рис. 3.7. Несколько векторов с одинаковыми координатами *x* и *y*, но разными координатами *z*

Для однозначной идентификации точки в трехмерном пространстве нужны три числа. Тройка чисел, таких как (4, 3, 5), называется координатами x, y и z трехмерного вектора. Как и прежде, их можно считать инструкциями по поиску нужной точки. Как показано на рис. 3.8, чтобы добраться до точки (4, 3, 5), сначала нужно сделать 4 шага в положительном направлении оси x, затем 3 шага в положительном направлении оси y и, наконец, 5 шагов в положительном направлении оси z.

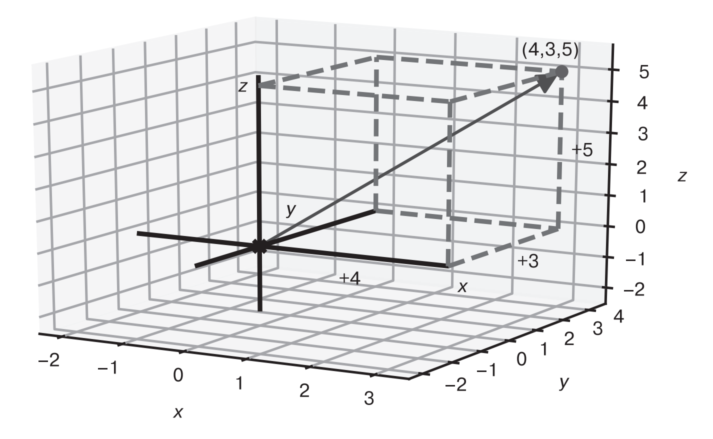


Рис. 3.8. Направление на точку в трехмерном пространстве задают координаты (4, 3, 5)

3.1.2. Рисование трехмерных изображений с помощью Python

Как и в предыдущей главе, для создания рисунков векторов в трехмерном пространстве я использую обертку на Python вокруг библиотеки Matplotlib. Ее реализацию можно найти в примерах с исходным кодом для этой книги, и далее я буду применять обертку, чтобы сосредоточиться на процессе рисования, а не на особенностях Matplotlib.

Моя обертка использует новые классы, такие как `Points3D` и `Arrow3D`, чтобы можно было отличить трехмерные объекты от их двухмерных аналогов. Новая функция `draw3d` знает, как интерпретировать и визуализировать эти объекты, чтобы они выглядели трехмерными. По умолчанию `draw3d()` рисует оси и начало координат, а также небольшой блок трехмерного пространства (рис. 3.9), даже если объекты для рисования не указаны.

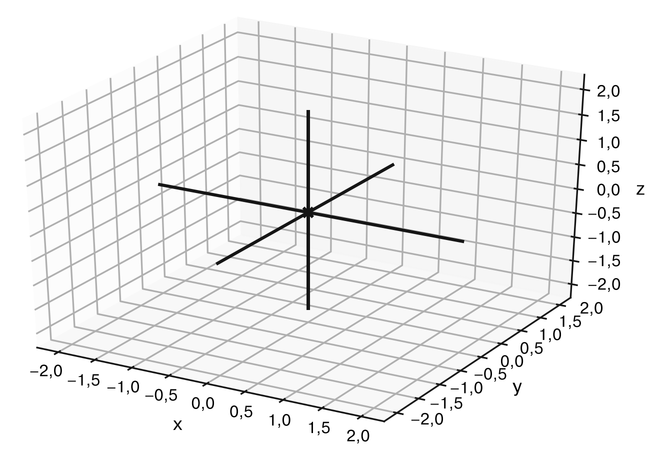


Рис. 3.9. Пустая область трехмерного пространства, нарисованная с помощью draw3d() и Matplotlib

Нарисованные оси x, y и z перпендикулярны, несмотря на то что на двухмерном рисунке они выглядят искаженными перспективой. Чтобы не загромождать рисунки, Matplotlib выводит единицы за границы блока, но начало координат и сами оси отображаются внутри. Началом является точка с координатами (0, 0, 0), а оси x, y и z исходят из нее в положительном и отрицательном направлениях.

Класс `Points3D` хранит набор векторов, которые мы рассматриваем как точки и которые поэтому отображаются как точки в трехмерном пространстве.

Например, следующий код нарисует векторы (2, 2, 2) и (1, –2, –2), как показано на рис. 3.10:

```
draw3d(
    Points3D((2,2,2),(1,-2,-2))
)
```

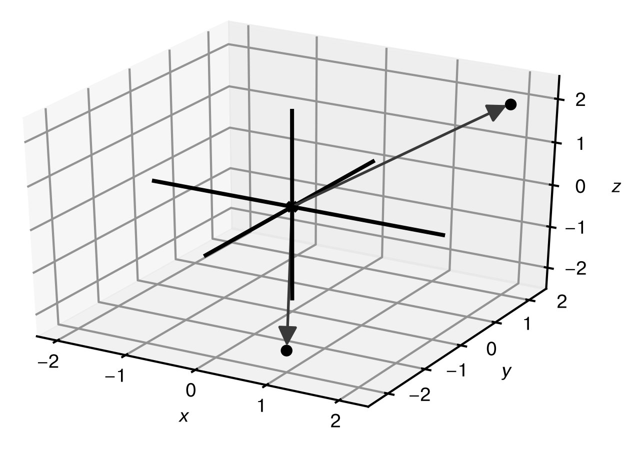


Рис. 3.10. Рисунок с точками (2, 2, 2) и (1, –2, –2)

Чтобы изобразить эти векторы в виде стрелок, их нужно представить как объекты `Arrow3D`. Можно также соединить кончики стрелок, добавив объект `Segment3D`, как показано далее, и получить рисунок, изображенный на рис. 3.11:

```
draw3d(
    Points3D((2,2,2),(1,-2,-2)),
    Arrow3D((2,2,2)),
    Arrow3D((1,-2,-2)),
    Segment3D((2,2,2), (1,-2,-2))
)
```

Глядя на рис. 3.11, немного сложно понять, в каком направлении указывают стрелки. Поэтому для большей ясности можно нарисовать пунктирные границы вокруг стрелок, чтобы они выглядели более объемными. Мы будем рисовать такие блоки так часто, что я определил класс `Box3D` для представления блока с одним углом в начале координат и противоположным углом в заданной точке. На рис. 3.12 показано, как выглядят такие блоки, но сначала вот код, который создал этот рисунок:

```
draw3d(
    Points3D((2,2,2),(1,-2,-2)),
    Arrow3D((2,2,2)),
```

```
    Arrow3D((1,-2,-2)),
    Segment3D((2,2,2), (1,-2,-2)),
    Box3D(2,2,2),
    Box3D(1,-2,-2)
)
```

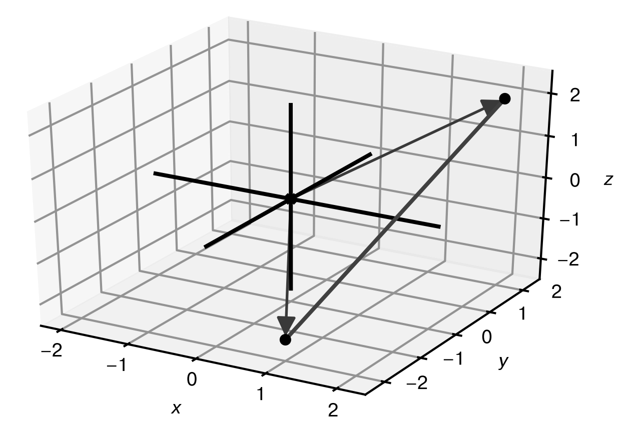


Рис. 3.11. Рисунок со стрелками

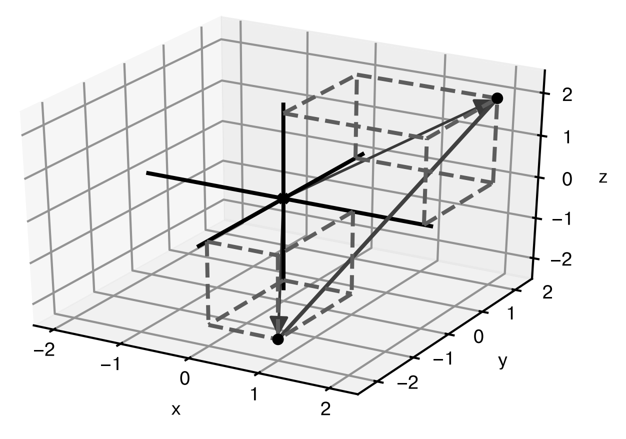


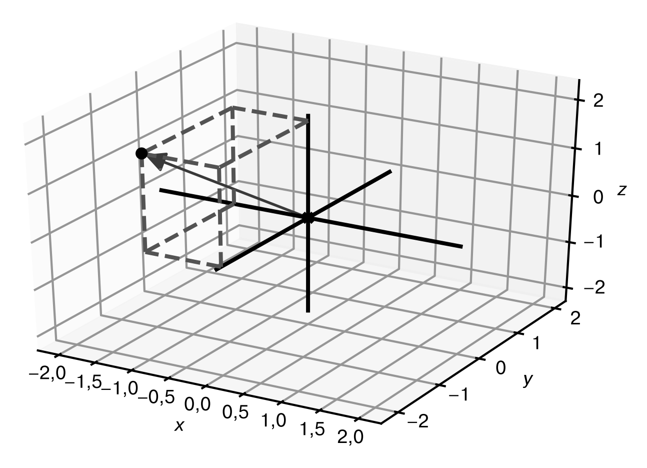
Рис. 3.12. Пунктирные рамки придают стрелкам дополнительный объем

В этой главе я использую именованные аргументы (имена которых, как я надеюсь, красноречиво объясняют их назначение), но не описываю их явно. Например, конструкторам большинства моих объектов можно передать именованный аргумент `color`, определяющий цвет объекта, отображаемого на рисунке.

3.1.3. Упражнения

Упражнение 3.1. Нарисуйте трехмерную стрелку и точку, представляющие координаты (–1, –2, 2), а также пунктирную рамку, придающую стрелке объем. Нарисуйте рисунок вручную, чтобы попрактиковаться, но далее будем использовать только Python.

Решение



Вектор (–1, –2, 2) и пунктирная рамка, придающая дополнительный объем

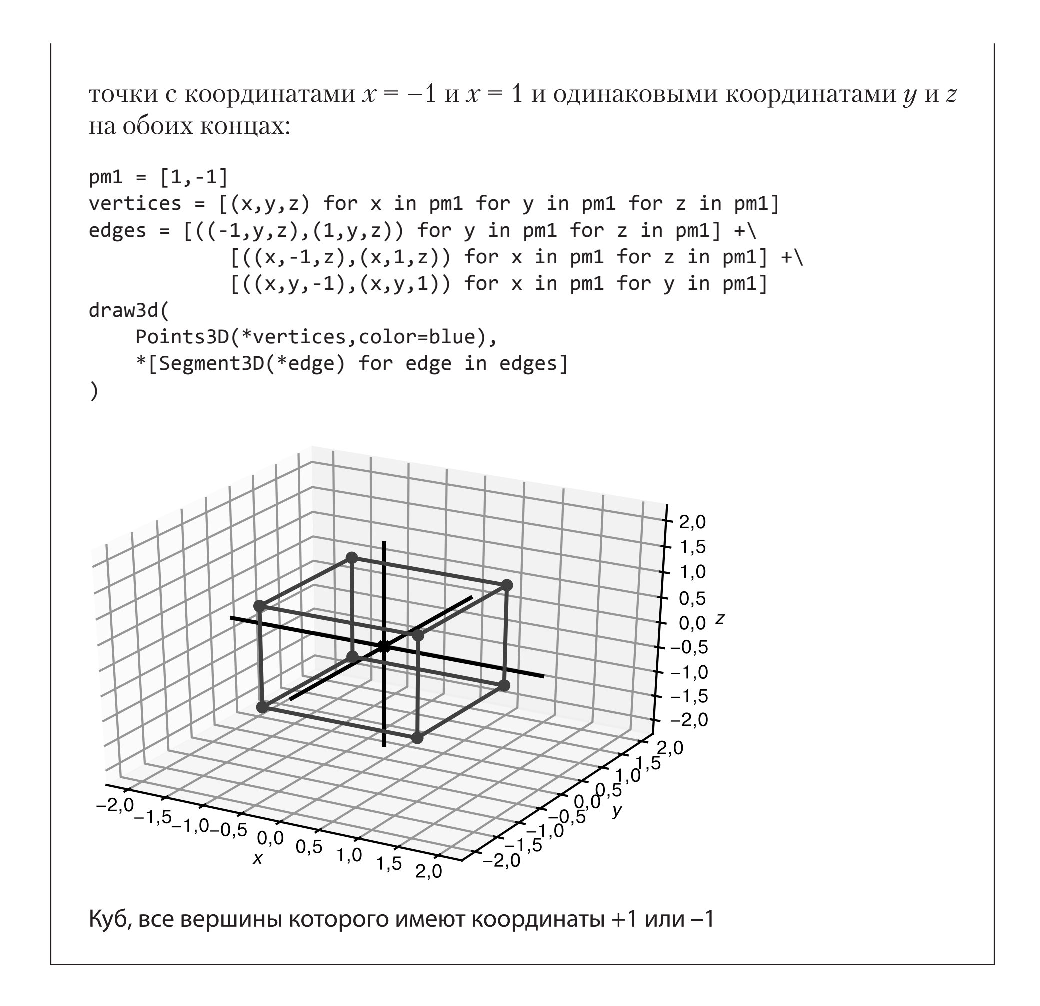
Упражнение 3.2. Мини-проект. Существует ровно восемь трехмерных векторов, все координаты которых равны +1 или –1. Например, (1, –1, 1) — один из них. Нарисуйте все восемь векторов как точки. Затем выясните, как соединить их отрезками, используя объекты Segment3D, чтобы сформировать контур куба.

Подсказка. Всего понадобится 12 сегментов.

Решение. Поскольку существует всего 8 вершин и 12 ребер, перечислить их все будет не слишком сложно, но я решил сделать это с помощью генератора списка. Для вершин я определил допустимые координаты x, y и z в виде списка [1, –1] и собрал все восемь результатов в список. Ребра я сгруппировал в три набора — в каждом по четыре ребра, параллельных одной из осей координат. Например, есть четыре ребра, соединяющих

точки с координатами $x = -1$ и $x = 1$ и одинаковыми координатами y и z на обоих концах:

```
pm1 = [1,-1]
vertices = [(x,y,z) for x in pm1 for y in pm1 for z in pm1]
edges = [((-1,y,z),(1,y,z)) for y in pm1 for z in pm1] +\
            [((x,-1,z),(x,1,z)) for x in pm1 for z in pm1] +\
            [((x,y,-1),(x,y,1)) for x in pm1 for y in pm1]
draw3d(
    Points3D(*vertices,color=blue),
    *[Segment3D(*edge) for edge in edges]
)
```



Куб, все вершины которого имеют координаты +1 или –1

3.2. АРИФМЕТИКА ТРЕХМЕРНЫХ ВЕКТОРОВ

Наличие готовых функций на Python позволяет легко визуализировать результаты арифметики трехмерных векторов. Все арифметические операции, которые мы видели при знакомстве с двухмерными векторами, имеют трехмерные аналоги, и они производят аналогичные геометрические эффекты.

3.2.1. Сложение трехмерных векторов

Сложение трехмерных векторов выполняется путем сложения соответствующих координат. Сложение векторов (2, 1, 1) и (1, 2, 2) дает в результате (2 + 1, 1 + 2, 1 + 2) = (3, 3, 3). Мы можем разместить эти два вектора друг за другом в любом

порядке, начиная с начала координат, чтобы добраться до точки векторной суммы (3, 3, 3), как показано на рис. 3.13.

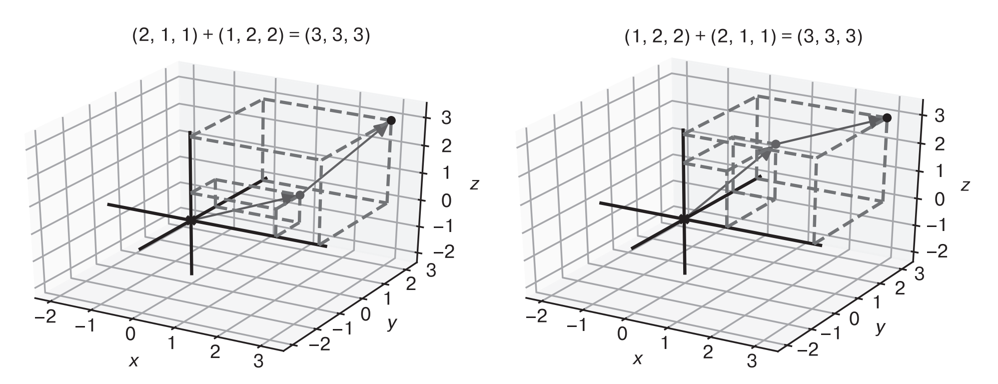


Рис. 3.13. Два визуальных примера сложения трехмерных векторов

По аналогии с двухмерными векторами можно сложить сколько угодно трехмерных векторов, суммируя по отдельности их координаты x, y и z. Эти три суммы дадут координаты нового вектора. Например, сложим три вектора: (1, 1, 3) + (2, 4, –4) + (4, 2, –2). Сумма их координат x 1, 2 и 4 равна 7. Сумма координат y также равна 7, а сумма координат z равна –3, соответственно, векторная сумма будет равна (7, 7, –3). Геометрическое представление суммы этих трех векторов показано на рис. 3.14.

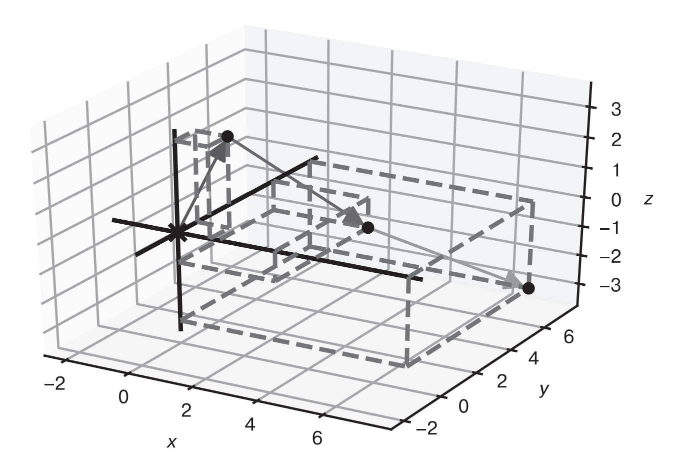


Рис. 3.14. Геометрическое представление суммы трех векторов

Мы можем написать на Python короткую функцию, складывающую любое количество векторов в двух или трех измерениях (или даже в большем количестве измерений, как вы увидите позже):

```
def add(*vectors):
    by_coordinate = zip(*vectors)
    coordinate_sums = [sum(coords) for coords in by_coordinate]
    return tuple(coordinate_sums)
```

Разберем ее работу. Функция `zip` извлечет из входных векторов их координаты x, y и z, например:

```
>>> list(zip(*[(1,1,3),(2,4,-4),(4,2,-2)]))
[(1, 2, 4), (1, 4, 2), (3, -4, -2)]
```

(Чтобы вывести результат, возвращаемый функцией `zip`, его нужно преобразовать в список.) Далее к каждой группе координат применяется функция `sum`, чтобы получить суммы координат x, y и z:

```
[sum(coords) for coords in [(1, 2, 4), (1, 4, 2), (3, -4, -2)]]
[7, 7, -3]
```

Наконец, полученный список преобразуется в кортеж, потому что до сих пор все векторы мы представляли в виде кортежей. Результат — это кортеж (7, 7, 3). Функцию `add` можно было бы уместить в одну строку (хотя такой код выглядит менее идиоматичным для Python):

```
def add(*vectors):
    return tuple(map(sum, zip(*vectors)))
```

3.2.2. Умножение трехмерных векторов на скаляр

Чтобы умножить трехмерный вектор на скаляр, нужно умножить каждую его координату на этот скаляр. Например, умножение вектора (1, 2, 3) на скаляр 2 дает в результате вектор (2, 4, 6). Этот вектор в два раза длиннее исходного, но указывает в том же направлении. На рис. 3.15 показан вектор $\mathbf{v} = (1, 2, 3)$ и результат его умножения на скаляр $2\mathbf{v} = (2, 4, 6)$.

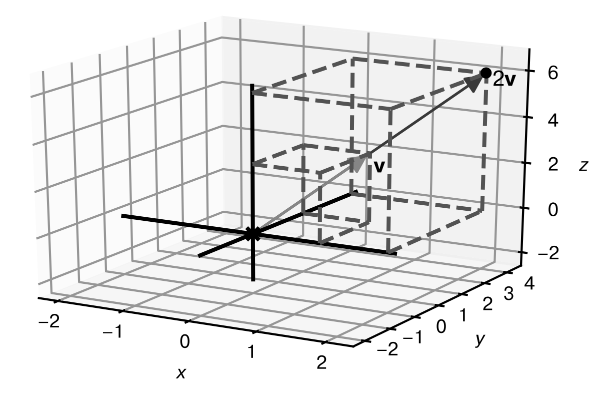


Рис. 3.15. Умножение вектора на скаляр 2 дает вектор, указывающий в том же направлении, но в два раза длиннее исходного вектора

3.2.3. Вычитание трехмерных векторов

В двухмерном пространстве разность двух векторов **v** – **w** — это вектор «от **w** до **v**», который называется *смещением*. В трехмерном пространстве все то же самое, то есть **v** – **w** — это смещение от **w** к **v**, или вектор, который можно прибавить к **w**, чтобы получить **v**. Если представить **v** и **w** как стрелки, начинающиеся в начале координат, то разность **v** – **w** — это стрелка, начинающаяся на конце **w** и заканчивающаяся на конце **v**. На рис. 3.16 показана разность векторов **v** = (–1, –3, 3) и **w** = (3, 2, 4) как в виде стрелки от **w** к **v**, так и в виде отдельной точки.

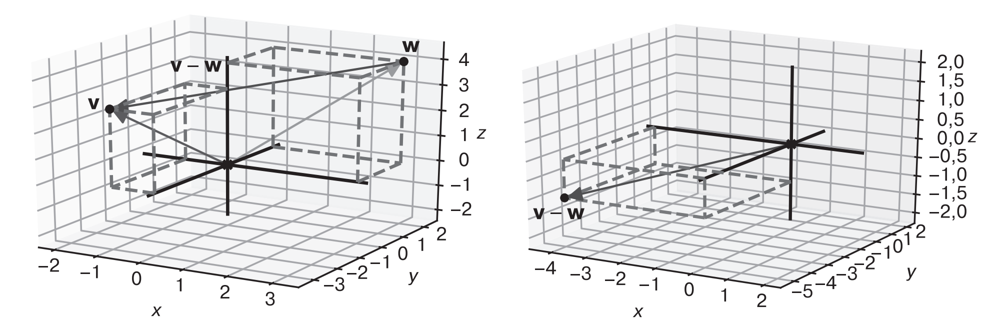


Рис. 3.16. Вычитание вектора **w** из вектора **v** дает смещение от **w** к **v**

Вычитание вектора **w** из вектора **v** выполняется путем вычитания соответствующих координат. Например, **v** – **w** дает в результате (–1 – 3, –3 – 2, 3 – 4) = = (–4, –5, –1). Эти координаты согласуются с изображением разности **v** – **w** на рис. 3.16, где показано, что получившийся вектор указывает в отрицательном направлении x, отрицательном направлении y и отрицательном направлении z.

Утверждая, что умножение на скаляр 2 делает вектор вдвое длиннее, я мыслю в терминах геометрического подобия. Если удвоить каждую из трех компонент **v**, что соответствует удвоению длины, ширины и глубины блока пространства, то расстояние по диагонали от одного угла до другого также должно удвоиться. Чтобы подтвердить или опровергнуть это утверждение, нужно знать, как вычисляются расстояния в трехмерном пространстве.

3.2.4. Вычисление длин и расстояний

В двухмерном пространстве мы вычисляли длину вектора по теореме Пифагора, используя тот факт, что вектор-стрелка и его компоненты образуют прямоугольный треугольник. Точно так же расстояние между двумя точками на плоскости совпадало с длиной вектора разности.

Если присмотреться внимательно, мы найдем в трехмерном пространстве все тот же прямоугольный треугольник, который поможет вычислить длину вектора.

Попробуем, например, найти длину вектора (4, 3, 12). Компоненты x и y по-прежнему дают нам прямоугольный треугольник, лежащий на плоскости, где $z = 0$. Гипотенуза этого треугольника имеет длину $\sqrt{\left(4^2 + 3^2\right)} = \sqrt{25} = 5$. Если бы это был двухмерный вектор, то мы на этом бы и закончили, но компонента z, равная 12, делает этот вектор немного длиннее (рис. 3.17).

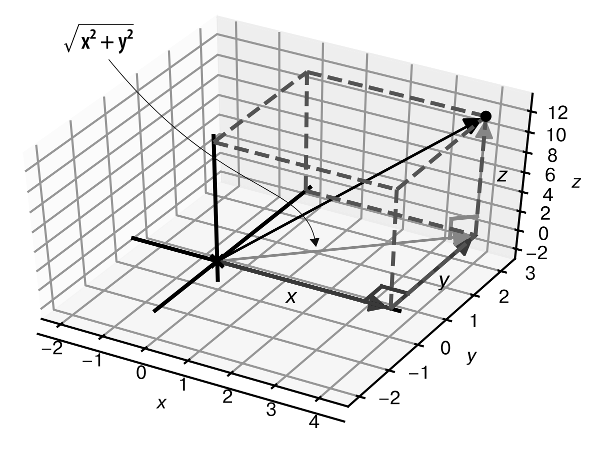


Рис. 3.17. Применение теоремы Пифагора для определения длины гипотенузы прямоугольного треугольника на плоскости *xy*

До сих пор все рассматривавшиеся нами векторы лежали на плоскости xy, где $z = 0$. Компонента x равна (4, 0, 0), компонента y равна (0, 3, 0), а их векторная сумма равна (4, 3, 0). Компонента z (0, 0, 12) перпендикулярна всем трем. Это полезное свойство, потому что дает второй прямоугольный треугольник, образованный векторами (4, 3, 0) и (0, 0, 12). Гипотенуза этого треугольника и есть исходный вектор (4, 3, 12), длину которого нужно найти. Сосредоточимся на этом втором прямоугольном треугольнике и найдем длину его гипотенузы (показана на рис. 3.18), снова воспользовавшись теоремой Пифагора.

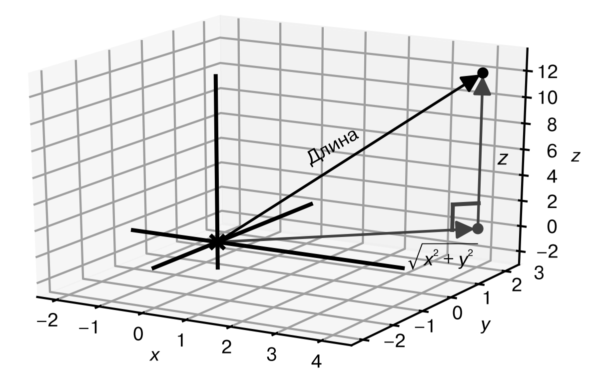


Рис. 3.18. Второе применение теоремы Пифагора дает длину трехмерного вектора

Возведение в квадрат длин двух известных сторон и извлечение квадратного корня из суммы квадратов должны дать искомую длину. В данном случае длины равны 5 и 12, поэтому результат $\sqrt{(5^2+12^2)}=13$. В общем случае длина трехмерного вектора вычисляется по следующей формуле:

$$\textbf{Длина} = \sqrt{\left(\sqrt{x^2+y^2}\right)^2+z^2} = \sqrt{x^2+y^2+z^2}.$$

Очень похоже на формулу вычисления длины в двухмерном пространстве. В обоих пространствах, двух- и трехмерном, длина вектора равна квадратному корню из суммы квадратов его компонентов. Поскольку в следующей функции длина входного кортежа нигде явно не указывается, она будет правильно работать и с двухмерными, и с трехмерными векторами:

```
from math import sqrt
    def length(v):
        return sqrt(sum([coord ** 2 for coord in v]))
```

Вызов, например, `length((3,4,12))` вернет 13.

3.2.5. Вычисление углов и направлений

По аналогии с двухмерным пространством трехмерные вектора можно представлять в виде стрелок, или смещений, определенной длины в определенном направлении. В двухмерном пространстве это означает, что для идентификации любого вектора достаточно двух чисел — длины и угла, образующих пару полярных координат. Чтобы указать направление в трехмерном пространстве, одного угла недостаточно, нужны два угла.

Чтобы получить первый угол, снова представим вектор без координаты z, как если бы он лежал на плоскости xy. Его также можно представить как тень, отбрасываемую при освещении источником света, расположенным очень высоко на оси z. Эта тень образует некоторый угол с положительным направлением оси x, аналогичный углу, который мы использовали в полярных координатах. Обозначим его греческой буквой ϕ (фи). Второй угол — это угол между вектором и осью z. Обозначим его греческой буквой θ (тета). Эти углы показаны на рис. 3.19.

Длина вектора, обозначим ее $\mathbf{r}$, вместе с углами ϕ и θ может описать любой вектор в трех измерениях. Вместе три числа, r, ϕ и θ, называются *сферическими координатами*, в отличие от декартовых координат x, y и z. Вычислить сферические координаты из декартовых можно с помощью тригонометрии, которую мы уже рассмотрели, поэтому не будем вдаваться в подробности. На самом деле в этой книге мы больше не станем использовать сферические координаты, но я хочу кратко сравнить их с полярными координатами.

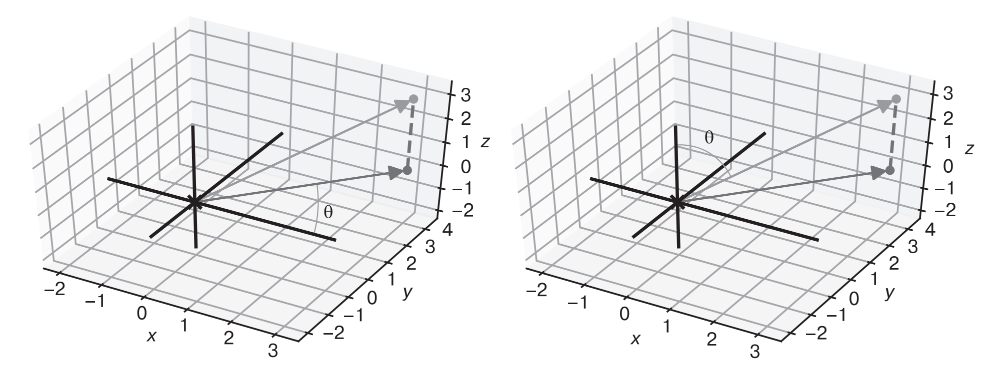


Рис. 3.19. Два угла, которые вместе задают направление трехмерного вектора

Полярные координаты были полезны, потому что позволяли выполнить любой поворот набора плоских векторов простым сложением или вычитанием углов, а также определить угол между двумя векторами, вычислив разность их углов в полярных координатах. В трехмерном пространстве ни один из углов ϕ и θ не позволяет сразу определить угол между двумя векторами. Мы могли бы поворачивать векторы вокруг оси z сложением или вычитанием с углом ϕ, но выполнять поворот вокруг любой другой оси в сферических координатах очень неудобно.

Нам нужны более универсальные инструменты для работы с углами и тригонометрией в трехмерном пространстве. В следующем разделе мы рассмотрим два таких инструмента, которые называются *векторными произведениями*.

3.2.6. Упражнения

Упражнение 3.3. Нарисуйте трехмерные векторы (4, 0, 3) и (–1, 0, 1) как объекты Arrow3D так, чтобы один начинался в конце другого в обоих порядках. Чему равна их векторная сумма?

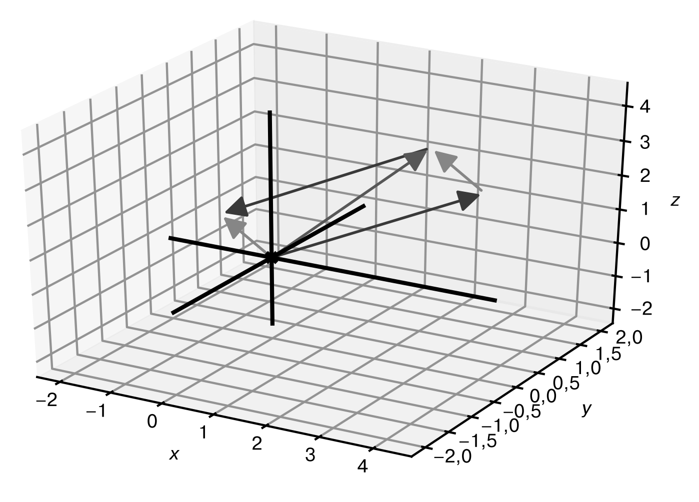
Решение. Векторную сумму можно найти с помощью созданной нами функции `add`:

```
>>> add((4,0,3),(-1,0,1))
(3, 0, 4)
```

Затем, чтобы нарисовать их цепочкой обоими способами, проводим стрелки от начала координат к каждой точке и затем от каждой точки — к векторной сумме (3, 0, 4). Подобно объекту Arrow, объект Arrow3D

принимает в первом аргументе конец стрелки и во втором необязательном аргументе — начало, если стрелка должна начинаться не в начале координат:

```
draw3d(
    Arrow3D((4,0,3),color=red),
    Arrow3D((-1,0,1),color=blue),
    Arrow3D((3,0,4),(4,0,3),color=blue),
    Arrow3D((-1,0,1),(3,0,4),color=red),
    Arrow3D((3,0,4),color=purple)
)
```



Как показывает результат сложения векторов, (4, 0, 3) + (–1, 0, 1) = (–1, 0, 1) + (4, 0, 3) = (3, 0, 4)

Упражнение 3.4. Представьте, что у нас есть два набора векторов: `vectors1 = [(1,2,3,4,5), (6,7,8,9,10)]` и `vectors2 = [(1,2), (3,4), (5,6)]`. Не прибегая к помощи Python, определите длину списка, который вернет `zip(*vectors1)` и `zip(*vectors2)`.

Решение. Первый вызов `zip` вернет список длиной 5. Поскольку в каждом из двух входных векторов пять координат, `zip(*vectors1)` вернет список с пятью кортежами, по два элемента в каждом. Аналогично, `zip(*vectors2)` вернет список длиной 2 — с двумя кортежами, содержащими все компоненты x и все компоненты y соответственно.

Упражнение 3.5. Мини-проект. Следующий генератор списков создает список с 24 векторами:

```
from math import sin, cos, pi
vs = [(sin(pi*t/6), cos(pi*t/6), 1.0/3) for t in range(0,24)]
```

Найдите сумму 24 векторов. Нарисуйте все 24 вектора цепочкой как объекты Arrow3D.

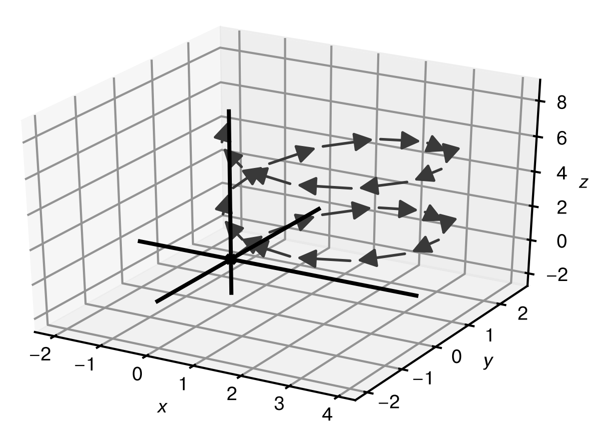
Решение. Если нарисовать все заданные векторы цепочкой, то получится спиралевидная форма:

```
from math import sin, cos, pi
vs = [(sin(pi*t/6), cos(pi*t/6), 1.0/3) for t in range(0,24)]

running_sum = (0,0,0)
arrows = []
for v in vs:
    next_sum = add(running_sum, v)
    arrows.append(Arrow3D(next_sum, running_sum))
    running_sum = next_sum
print(running_sum)
draw3d(*arrows)
```

Начальные значения текущей суммы цепочки векторов

Чтобы нарисовать следующий вектор в цепочке, прибавляем его к текущей сумме и соединяем стрелкой предыдущую текущую сумму со следующей



Определение суммы 24 векторов в трехмерном пространстве

Сумма:

```
(-4.440892098500626e-16, -7.771561172376096e-16, 7.9999999999999964)
```

или, если округлить, (0, 0, 8).

Упражнение 3.6. Напишите функцию `scale(scalar, vector)`, которая возвращает результат умножения скаляра `scalar` на вектор `vector`. В частности, напишите ее так, чтобы она могла принимать и двухмерные, и трехмерные векторы, а также векторы с любым другим числом измерений.

Решение. Умножение реализуется с помощью генератора списков: он умножает каждую координату в векторе на скаляр. Затем генератор преобразуется в кортеж:

```
def scale(scalar,v):
    return tuple(scalar * coord for coord in v)
```

Упражнение 3.7. Пусть **u** = (1, –1, –1) и **v** = (0, 0, 2). Найдите результат **u** + (1/2)(**v** – **u**)?

Решение. Имея **u** = (1, –1, –1) и **v** = (0, 0, 2), можно сначала вычислить (**v** – **u**) = (0 – 1, 0 – (–1), 2 – (–1)) = (–1, 1, 3). Далее, (1/2)(**v** – **u**) будет равно (–1/2, 1/2, 3/2). Конечный желаемый результат **u** + (1/2)(**v** – **u**) будет равен (1/2, –1/2, 1/2). Между прочим, эта точка находится ровно посередине между точками **u** и **v**.

Упражнение 3.8. Попробуйте найти ответы на вопросы в этом упражнении вручную, а затем проверьте их с помощью кода на Python. Какова длина двухмерного вектора (1, 1)? Какова длина трехмерного вектора (1, 1, 1)? Мы еще не говорили о четырехмерных векторах — у них четыре координаты. Попробуйте определить длину четырехмерного вектора с координатами (1, 1, 1, 1).

Решение. Длина вектора (1, 1) равна $\sqrt{1^2+1^2}=\sqrt{2}$. Длина вектора (1, 1, 1) равна $\sqrt{1^2+1^2+1^2}=\sqrt{2}$. Как вы наверняка догадались, длина векторов большей размерности вычисляется по той же формуле. Длина вектора (1, 1, 1, 1) равна $\sqrt{1^2+1^2+1^2+1^2}=\sqrt{4}$, то есть 2.

Упражнение 3.9. Мини-проект. Координаты 3, 4, 12 независимо от порядка создают вектор длиной, равной целому числу 13. Это довольно необычно, потому что большинство чисел не являются идеальными квадратами и квадратный корень в формуле длины обычно возвращает иррациональное число. Найдите другую тройку целых чисел, определяющую координаты вектора с целочисленной длиной.

Решение. Следующий код ищет тройки убывающих целых чисел меньше 100 (это число выбрано произвольно):

```
def vectors_with_whole_number_length(max_coord=100):
    for x in range(1,max_coord):
        for y in range(1,x+1):
            for z in range(1,y+1):
                if length((x,y,z)).is_integer():
                    yield (x,y,z)
```

Эта функция нашла 869 векторов с целочисленными координатами и длинами. Самый короткий из них — (2, 2, 1), его длина 3, а самый длинный — (99, 90, 70) с длиной 150.

Упражнение 3.10. Найдите вектор, указывающий в том же направлении, что и (–1, –1, 2), но с длиной 1.

Подсказка. Найдите подходящий скаляр, чтобы умножить на него исходный вектор и получить требуемую длину.

Решение. Вектор (–1, –1, 2) имеет длину около 2,45, поэтому искомый скаляр равен 1/2,45. Умножение исходного вектора на него даст вектор с длиной 1:

```
>>> length((-1,-1,2))
2.449489742783178
>>> s = 1/length((-1,-1,2))
>>> scale(s,(-1,-1,2))
(-0.4082482904638631, -0.4082482904638631, 0.8164965809277261)
>>> length(scale(s,(-1,-1,2)))
1.0
```

Округлив каждую координату до сотых долей, получаем искомый вектор (–0,41, –0,41, 0,82).

3.3. СКАЛЯРНОЕ ПРОИЗВЕДЕНИЕ ВЕКТОРОВ: МЕРА СОНАПРАВЛЕННОСТИ ВЕКТОРОВ

Один из видов умножения, который мы уже видели для векторов, — умножение на скаляр — объединяет скаляр (действительное число) и вектор для получения нового вектора. Но мы пока не говорили ни о каких способах умножения одного вектора на другой. Оказывается, есть два способа сделать это, и оба имеют важный геометрический смысл. Первый называется *скалярным произведением* и записывается с помощью оператора точки (например, $\mathbf{u} \cdot \mathbf{v}$), а второй называется *векторным* произведением (например, $\mathbf{u} \times \mathbf{v}$). Для чисел эти два символа операторов означают одно и то же, например, $3 \cdot 4 = 3 \times 4$, но для векторов операции $\mathbf{u} \cdot \mathbf{v}$ и $\mathbf{u} \times \mathbf{v}$ не только различаются обозначением, но и имеют совершенно разный смысл.

Скалярное произведение двух векторов дает в результате скаляр (число), а векторное произведение двух векторов дает другой вектор. Однако обе операции помогают рассуждать о длинах и направлениях векторов в трехмерном пространстве. Но пойдем по порядку и первым рассмотрим скалярное произведение.

3.3.1. Изображение скалярного произведения

Скалярное произведение, также называемое *внутренним произведением*, — это операция над двумя векторами, которая возвращает скаляр. Иначе говоря, принимая два вектора, $\mathbf{u}$ и $\mathbf{v}$, операция $\mathbf{u} \cdot \mathbf{v}$ дает действительное число. Скалярное произведение может применяться и к двухмерным, и к трехмерным векторам, а также к векторам с любым количеством измерений. Скалярное произведение можно рассматривать как оценку сонаправленности пары векторов. Сначала рассмотрим несколько векторов на плоскости *xy* и их скалярные произведения, чтобы получить некоторое представление о том, как работает эта операция.

Векторы $\mathbf{u}$ и $\mathbf{v}$ имеют длину 4 и 5 соответственно и указывают почти в одном направлении. Их скалярное произведение — положительное число, а это означает, что они сонаправлены (рис. 3.20).

Два вектора, указывающих в близких направлениях, имеют положительное скалярное произведение, и чем длиннее вектор, тем больше произведение. Короткие векторы, имеющие близкие направления, дают меньшее, но все же положительное скалярное произведение. На рис. 3.21 показаны два других вектора $\mathbf{u}$ и $\mathbf{v}$ с длиной 2.

Напротив, если векторы указывают в противоположных или почти противоположных направлениях, их скалярное произведение отрицательно (рис. 3.22 и 3.23). Чем больше величина векторов, тем больше их отрицательное скалярное произведение.

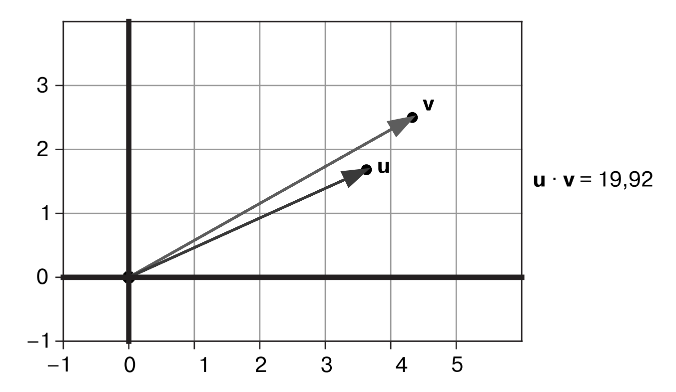


Рис. 3.20. Два вектора указывают в близких направлениях, благодаря чему скалярное произведение дает большое положительное число



Рис. 3.21. Два коротких вектора, указывающих в близких направлениях, дают маленькое, но все же положительное скалярное произведение

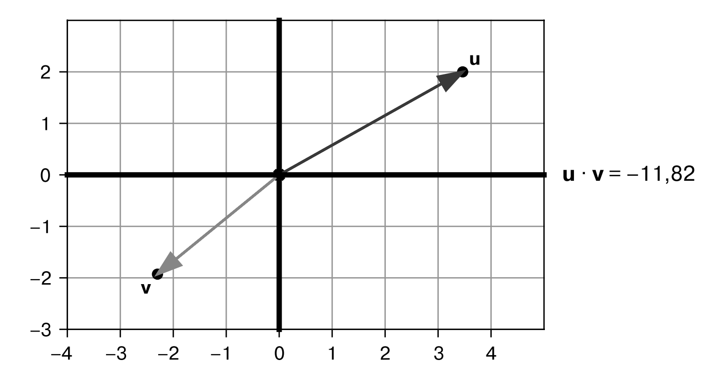


Рис. 3.22. Векторы, указывающие в противоположных направлениях, дают отрицательное скалярное произведение

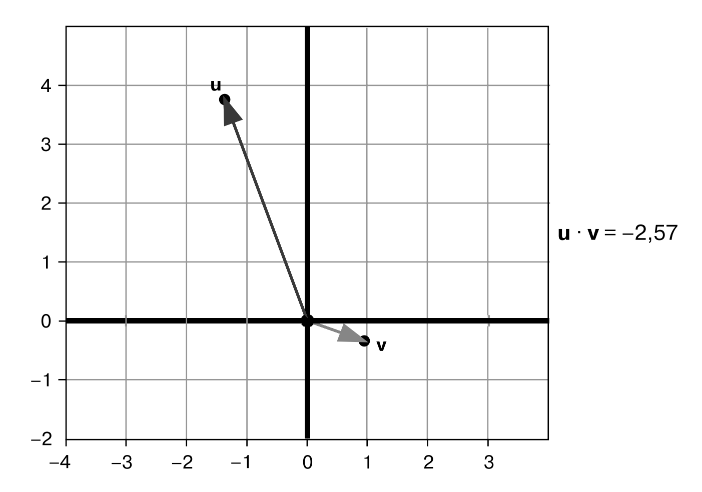


Рис. 3.23. Два коротких вектора, указывающих в противоположных направлениях, дают меньшее по величине, но все же отрицательное скалярное произведение

Не все пары векторов четко указывают в близких или противоположных направлениях, и скалярное произведение обнаруживает это. Как показано на рис. 3.24, если два вектора направлены строго перпендикулярно, их скалярное произведение равно нулю независимо от их длин.

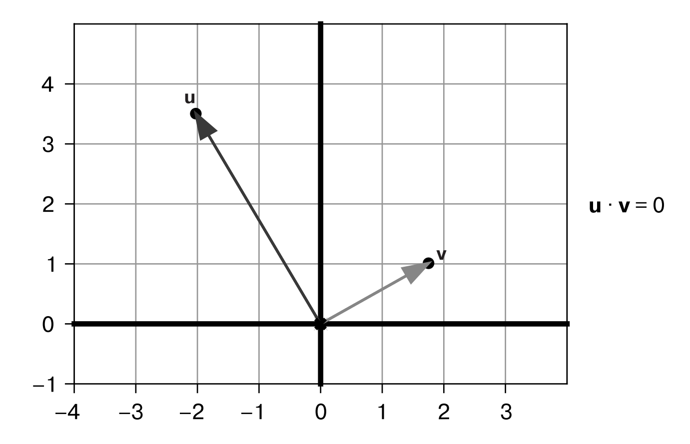


Рис. 3.24. Скалярное произведение перпендикулярных векторов равно нулю

Как оказывается, это одно из самых важных применений скалярного произведения: оно позволяет определить, перпендикулярны ли два вектора, не прибегая к тригонометрии. Признак перпендикулярности служит также разделителем двух других случаев: если угол между векторами меньше 90°, то скалярное

произведение этих векторов положительное. Если угол больше 90°, то скалярное произведение отрицательное. Я еще не рассказал вам, как вычислить скалярное произведение, но теперь вы знаете, как интерпретировать его значение. Перейдем к вычислению.

3.3.2. Вычисление скалярного произведения

Существует простая формула вычисления скалярного произведения: нужно перемножить соответствующие координаты, а затем сложить произведения. Например, в скалярном произведении (1, 2, –1) · (3, 0, 3) результат умножения координат x двух векторов равен 3, результат умножения координат y равен 0 и результат умножения координат z равен –3. Соответственно, сумма 3 + 0 + (–3) = 0, и результат самого скалярного произведения равен нулю. Если мои предыдущие утверждения верны, то эти два вектора должны быть перпендикулярны. Их изображение (рис. 3.25) подтверждает это, если смотреть с правильного ракурса!

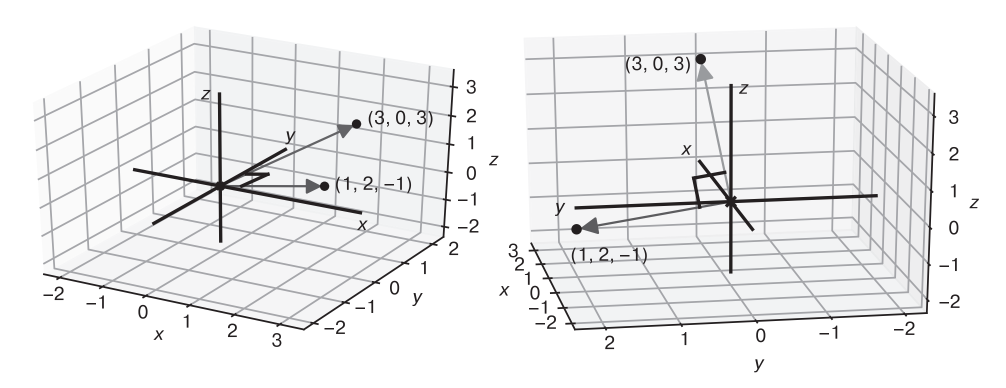
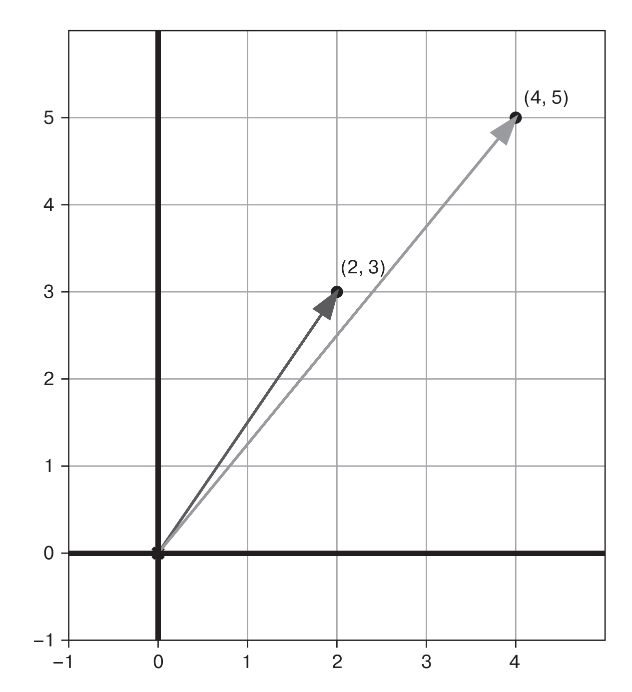


Рис. 3.25. Два трехмерных вектора, чье скалярное произведение равно нулю, действительно перпендикулярны

Чувство перспективы в трехмерном пространстве может вводить нас в заблуждение, что делает еще более полезной возможность *вычислять* относительные направления, а не оценивать их на глаз. В качестве еще одного примера на рис. 3.26 показано, что двухмерные векторы (2, 3) и (4, 5) имеют близкие направления в плоскости xy. Произведение координат x — 2 · 4 = 8, а произведение координат y — 3 · 5 = 15. Сумма, она же результат скалярного произведения, составляет 8 + 15 = 23. Будучи положительным числом, этот результат подтверждает, что угол между векторами меньше 90°. Эти векторы имеют одинаковую относительную геометрию, как бы мы их ни рассматривали, в двух измерениях или в трех — как векторы (2, 3, 0) и (4, 5, 0), которые лежат на плоскости с координатой z = 0.



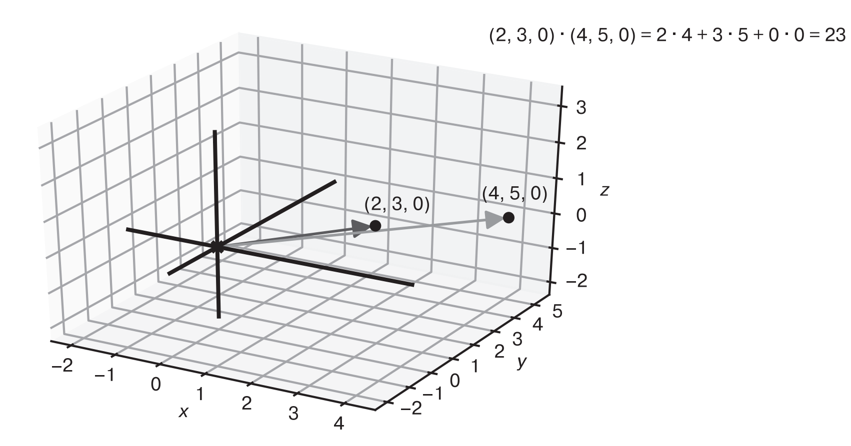


Рис. 3.26. Еще один пример вычисления скалярного произведения

Мы можем написать на Python функцию скалярного произведения, которая принимает любые пары векторов и вычисляет скалярное произведение, если они имеют одинаковое количество координат, например:

```
def dot(u,v):
    return sum([coord1 * coord2 for coord1,coord2 in zip(u,v)])
```

Этот код использует функцию `zip` для объединения соответствующих координат в кортежи, а затем умножает каждую пару и складывает произведения. Задействуем эту функцию для дальнейшего изучения поведения скалярного произведения.

3.3.3. Примеры скалярных произведений

Неудивительно, что скалярное произведение двух векторов, лежащих на разных осях, равно нулю. Мы знаем, что оси, а значит, и сами векторы перпендикулярны:

```
>>> dot((1,0),(0,2))
0
>>> dot((0,3,0),(0,0,-5))
0
```

Мы также можем подтвердить, что более длинные векторы дают большие скалярные произведения. Например, умножение любого вектора на 2 удваивает результат скалярного произведения:

```
>>> dot((3,4),(2,3))
18
>>> dot(scale(2,(3,4)),(2,3))
36
>>> dot((3,4),scale(2,(2,3)))
36
```

Оказывается, скалярное произведение пропорционально длинам обоих векторов. Если взять скалярное произведение двух векторов, указывающих в одном направлении, то оно будет точно равно произведению длин. Например, (4, 3) имеет длину 5, а (8, 6) — длину 10. Скалярное произведение равно $5 \cdot 10$:

```
>>> dot((4,3),(8,6))
50
```

Конечно, скалярное произведение не всегда равно произведению длин своих входов. Векторы (5, 0), (–3, 4), (0, –5) и (–4, –3) имеют одинаковую длину 5, но дают разные скалярные произведения с вектором (4, 3), как показано на рис. 3.27.

Скалярное произведение двух векторов с длиной 5 находится в диапазоне от $5 \cdot 5 = 25$, когда они указывают строго в одном направлении и до –25, когда указывают в противоположных направлениях. В следующем наборе упражнений я предложу вам самим убедиться, что скалярное произведение двух векторов может варьироваться от произведения их длин до произведения длин с отрицательным знаком.

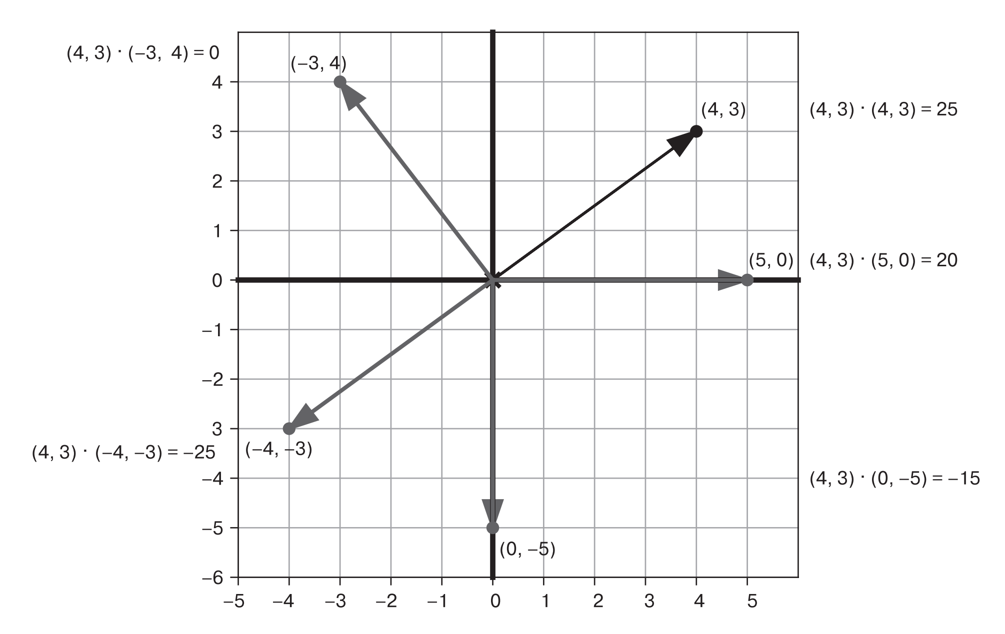


Рис. 3.27. Векторы одной и той же длины могут давать разные скалярные произведения с вектором (4, 3) в зависимости от их направления

3.3.4. Измерение углов с помощью скалярного произведения

Мы уже видели, что результат скалярного произведения зависит от угла между векторами. В частности, скалярное произведение $\mathbf{u} \cdot \mathbf{v}$ находится в диапазоне от 1 до −1 произведения длин **u** и **v** при изменении угла от 0 до 180°. Видели также функцию, которая ведет себя подобным образом, — функцию косинуса. Как оказывается, у скалярного произведения есть альтернативная формула. Если $|\mathbf{u}|$ и $|\mathbf{v}|$ обозначают длины векторов **u** и **v**, то скалярное произведение определяется выражением

$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}| \cdot |\mathbf{v}| \cdot \cos \theta,$$

где θ — угол между векторами **u** и **v**. В принципе, это дает нам новый способ вычисления скалярного произведения: измерить длины двух векторов и угол между ними, чтобы получить результат. Предположим, у нас есть два вектора с известными длинами 3 и 2 и с помощью транспортира мы определили, что угол между ними равен 75° (рис. 3.28).

Скалярное произведение двух векторов на рис. 3.28 равно $3 \cdot 2 \cdot \cos 75°$. Преобразовав угол из градусов в радианы, результат, примерно равный 1,55, можно вычислить с помощью Python:

```
>>> from math import cos,pi
>>> 3 * 2 * cos(75 * pi / 180)
1.5529142706151244
```

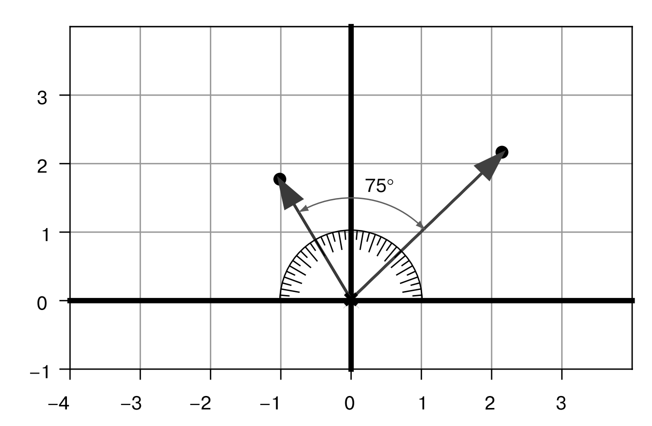


Рис. 3.28. Два вектора с длинами 3 и 2, образующие угол 75°

При выполнении вычислений с векторами обычно начинают с координат и вычисляют по ним углы. Мы можем объединить формулы, чтобы получить угол через координаты: сначала нужно вычислить скалярное произведение и длины векторов, а затем найти угол.

Например, найдем угол между векторами (3, 4) и (4, 3). Их скалярное произведение равно 24, и оба имеют длину 5. Согласно новой формуле скалярного произведения

$$(3, 4) \cdot (4, 3) = 24 = 5 \cdot 5 \cdot \cos\theta = 25 \cos\theta.$$

Из тождества $24 = 25 \cos\theta$ следует, что $\cos\theta = 24/25$. Используя функцию `math.acos`, находим значение θ, равное 0,284 рад, или 16,3°.

Этот пример показывает, почему в двухмерном пространстве нет необходимости вычислять скалярное произведение. В главе 2 мы видели, как вычислить угол между вектором и положительным направлением оси x. Творчески применяя эту формулу, можно найти абсолютно любой угол на плоскости. Скалярное произведение становится по-настоящему необходимым в трехмерном пространстве, где отношение координат почти ничем не может нам помочь.

Например, мы можем применить ту же формулу, чтобы найти угол между (1, 2, 2) и (2, 2, 1). Скалярное произведение $1 \cdot 2 + 2 \cdot 2 + 2 \cdot 1 = 8$, а длины обоих векторов равны 3. Это означает, что $8 = 3 \cdot 3 \cdot \cos\theta$, откуда $\cos\theta = 8/9$ и $\theta = 0{,}476$ рад, или 27,3°.

Этот процесс одинаков и для двухмерного, и для трехмерного пространства, и мы будем использовать его снова и снова. Мы можем сэкономить свои силы, написав на Python функцию вычисления угла между двумя векторами. Поскольку

ни функция dot, ни функция length не ограничены определенным количеством измерений, то и новая функция получится универсальной. Мы можем использовать тот факт, что $\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}| \cdot |\mathbf{v}| \cdot \cos\theta$ и, следовательно,

$$\cos\theta = \frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{u}| \cdot |\mathbf{v}|}$$

и

$$\theta = \arccos\left(\frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{u}| \cdot |\mathbf{v}|}\right).$$

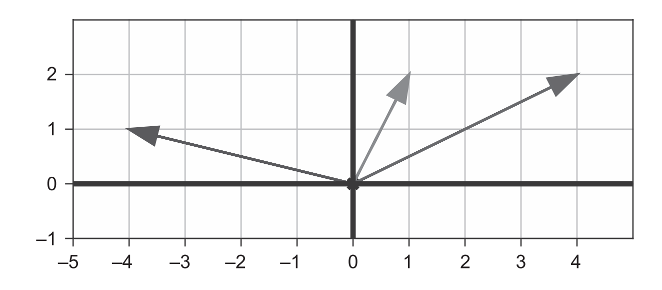
Эту формулу легко выразить в коде на Python:

```
def angle_between(v1,v2):
    return acos(
                dot(v1,v2) /
                (length(v1) * length(v2))
          )
```

Этот код никак не зависит от количества измерений векторов $\mathbf{v}_1$ и $\mathbf{v}_2$. Векторы могут быть представлены кортежами и с двумя, и с тремя координатами (и даже с четырьмя или большим числом координат, как будет обсуждаться в следующих главах). Однако векторное произведение, которое мы рассмотрим далее, можно вычислить только в трех измерениях.

3.3.5. Упражнения

Упражнение 3.11. На основе следующего рисунка расположите $\mathbf{u} \cdot \mathbf{v}$, $\mathbf{u} \cdot \mathbf{w}$ и $\mathbf{v} \cdot \mathbf{w}$ в порядке уменьшения.



Решение. Произведение $\mathbf{u} \cdot \mathbf{v}$ — единственное положительное скалярное произведение, потому что $\mathbf{u}$ и $\mathbf{v}$ — единственная пара векторов, угол между которыми меньше прямого. Кроме того, $\mathbf{u} \cdot \mathbf{w}$ меньше (имеет большее по абсолютной величине отрицательное значение), чем $\mathbf{v} \cdot \mathbf{w}$, потому что $\mathbf{u}$ больше и дальше от $\mathbf{w}$, соответственно $\mathbf{u} \cdot \mathbf{v} > \mathbf{v} \cdot \mathbf{w} > \mathbf{u} \cdot \mathbf{w}$.

Упражнение 3.12. Найдите скалярное произведение между векторами $(-1, -1, 1)$ и $(1, 2, 1)$. Угол между этими двумя трехмерными векторами больше, меньше или равен $90°$?

Решение. Скалярное произведение векторов $(-1, -1, 1)$ и $(1, 2, 1)$ составляет $-1 \cdot 1 + -1 \cdot 2 + 1 \cdot 1 = -2$. Поскольку результат отрицательный, эти векторы образуют угол больше $90°$.

Упражнение 3.13. Мини-проект. Для двух трехмерных векторов **u** и **v** значения $(2\mathbf{u}) \cdot \mathbf{v}$ и $\mathbf{u} \cdot (2\mathbf{v})$ равны $2(\mathbf{u} \cdot \mathbf{v})$. В данном случае $\mathbf{u} \cdot \mathbf{v} = 18$, а оба выражения — $(2\mathbf{u}) \cdot \mathbf{v}$ и $\mathbf{u} \cdot (2\mathbf{v})$ — дают одинаковый результат 36, вдвое превышающий результат произведения $\mathbf{u} \cdot \mathbf{v}$. Покажите, что это верно для любого действительного числа s, а не только для 2. Иначе говоря, покажите, что для любого s значения $(s\mathbf{u}) \cdot \mathbf{v}$ и $\mathbf{u} \cdot (s\mathbf{v})$ равны $s(\mathbf{u} \cdot \mathbf{v})$.

Решение. Обозначим координаты **u** и **v**, например, как $\mathbf{u} = (a, b, c)$ и $\mathbf{v} = (d, e, f)$. Тогда $\mathbf{u} \cdot \mathbf{v} = ad + be + cf$. Поскольку $s\mathbf{u} = (sa, sb, sc)$ и $s\mathbf{v} = (sd, se, sf)$, можно показать оба результата, разложив скалярные произведения. Вот доказательство того, что умножение на скаляр масштабирует результат скалярного произведения:

$$
\begin{aligned}
(s\mathbf{u}) \cdot \mathbf{v} &= (sa, sb, sc) \cdot (\mathrm{d}, e, f) = \\
&= sad + sbe + scf = \\
&= s(ad + be + cf) = \\
&= s(\mathbf{u} \cdot \mathbf{v}).
\end{aligned}
$$

Доказательство, что умножение на скаляр масштабирует результат скалярного произведения

То же верно для другого скалярного произведения:

$$
\begin{aligned}
\mathbf{u} \cdot (s\mathbf{v}) &= (a, b, c) \cdot (sd, se, sf) = \\
&= asd + bse + csf = \\
&= s(ad + be + cf) = \\
&= s\,(\mathbf{u} \cdot \mathbf{v}).
\end{aligned}
$$

Доказательство, что то же верно для второго скалярного произведения

Упражнение 3.14. Мини-проект. Объясните алгебраически, почему скалярное произведение вектора с самим собой равно квадрату его длины.

Решение. Пусть вектор имеет координаты (a, b, c), тогда скалярное произведение с самим собой будет $a \cdot a + b \cdot b + c \cdot c$, а его длина составит $\sqrt{a \cdot a + b \cdot b + c \cdot c}$, то есть действительно квадрат.

Упражнение 3.15. Мини-проект. Найдите вектор **u** длиной 3 и вектор **v** длиной 7 такие, что $\mathbf{u} \cdot \mathbf{v} = 21$. Найдите другую пару векторов **u** и **v**, такую, что $\mathbf{u} \cdot \mathbf{v} = -21$. Наконец, найдите еще три пары векторов длиной 3 и 7 и покажите, что их скалярные произведения лежат в диапазоне между –21 и 21.

Решение. Два вектора, указывающие точно в одном направлении (например, лежащие на оси x), будут иметь максимально возможное скалярное произведение:

```
>>> dot((3,0),(7,0))
21
```

Два вектора, указывающие точно в противоположных направлениях (например, в положительном и отрицательном направлениях оси y), будут иметь наименьшее из возможных скалярное произведение:

```
>>> dot((0,3),(0,-7))
-21
```

Используя полярные координаты, можно легко сгенерировать еще несколько векторов с длинами 3 и 7, образующих случайные углы:

```
from vectors import to_cartesian
from random import random
from math import pi

def random_vector_of_length(l):
    return to_cartesian((l, 2*pi*random()))

pairs = [(random_vector_of_length(3), random_vector_of_length(7))
             for i in range(0,3)]
for u,v in pairs:
    print("u = %s, v = %s" % (u,v))
    print("length of u: %f, length of v: %f, dot product :%f" %
              (length(u), length(v), dot(u,v)))
```

Упражнение 3.16. Пусть **u** и **v** — векторы такие, что |**u**| = 3,61 и |**v**| = 1,44. Пусть угол между **u** и **v** равен 101,3°. Определите, чему равно **u** · **v**. Варианты ответа:

1. 5,198.
2. 5,098.
3. −1,019.
4. 1,019.

Решение. Заданные значения можно подставить в новую формулу скалярного произведения и с соответствующим преобразованием градусов в радианы оценить результат с помощью Python:

```
>>> 3.61 * 1.44 * cos(101.3 * pi / 180)
-1.0186064362303022
```

После округления до трех знаков после запятой этот результат соответствует варианту 3.

Упражнение 3.17. Мини-проект. Найдите угол между векторами (3, 4) и (4, 3), преобразовав их в полярные координаты и вычислив разность углов. Варианты ответа:

1. 1,569.
2. 0,927.
3. 0,643.
4. 0,284.

Подсказка. Вариант должен совпадать со значением скалярного произведения.

Решение. Вектор (3, 4) образует больший угол с положительным направлением оси x, поэтому вычтем угол (4, 3) из угла (3, 4). Результат точно соответствует варианту 4:

```
>>> from vectors import to_polar
>>> r1,t1 = to_polar((4,3))
>>> r2,t2 = to_polar((3,4))
>>> t1-t2
-0.2837941092083278
>>> t2-t1
0.2837941092083278
```

Упражнение 3.18. Чему равен угол между (1, 1, 1) и (–1, –1, 1) в градусах? Варианты ответа:

1. 180°.
2. 120°.
3. 109,5°.
4. 90°.

Решение. Длины обоих векторов равны $\sqrt{3}$, или примерно 1,732. Их скалярное произведение $1 \cdot (-1) + 1 \cdot (-1) + 1 \cdot 1 = -1$, соответственно, $-1 = \sqrt{3} \cdot \sqrt{3} \cdot \cos\theta$, откуда получаем $\cos \theta = -1/3$. То есть угол примерно равен 1,911 рад, или 109,5° (вариант 3).

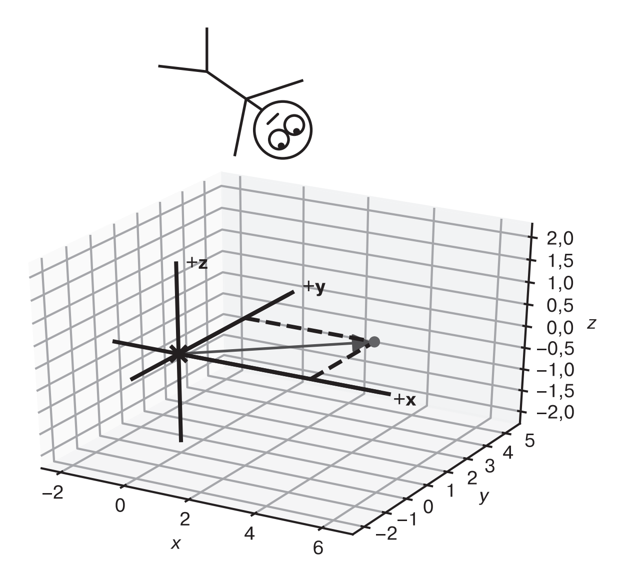
3.4. ВЕКТОРНОЕ ПРОИЗВЕДЕНИЕ: МЕРА ОРИЕНТИРОВАННОЙ ПЛОЩАДИ

Как говорилось ранее, векторное произведение двух трехмерных векторов $\mathbf{u} \times \mathbf{v}$ дает другой трехмерный вектор. Векторное произведение, как и скалярное, дает результат, зависящий от длин и относительных направлений исходных векторов, но при этом результат определяет не только некоторую величину, но еще и направление. Чтобы вы могли понять суть векторного произведения, мы должны поговорить о направлении в трехмерном пространстве.

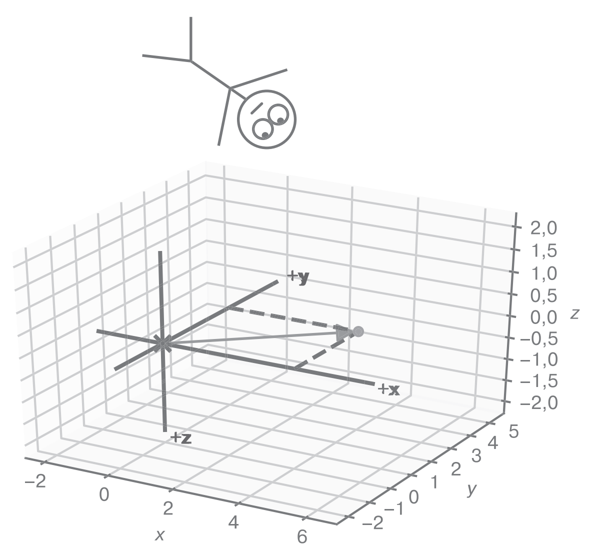
3.4.1. Ориентация в трехмерном пространстве

Представляя в начале главы оси *x*, *y* и *z*, я сделал два утверждения. Во-первых, что знакомая нам плоскость *xy* существует в трехмерном мире. Во-вторых, что ось *z* перпендикулярна плоскости *xy* и последняя находится на отметке $z = 0$. О чем я не заявил четко и ясно, так это о том, что положительное направление оси *z* указывает вверх, а не вниз.

Другими словами, если посмотреть на плоскость *xy* с обычной точки зрения, то положительный конец оси *z*, выходящей из плоскости, будет направлен на нас. Можно также представить, что положительный конец оси *z* направлен от нас (рис. 3.29).



Положительный конец оси z направлен к нам



Положительный конец оси z направлен от нас

Рис. 3.29. Располагаемся в трехмерном пространстве так, чтобы видеть плоскость *xy* так же, как видели ее в главе 2. Выбираем такое направление оси *z*, чтобы положительный конец был направлен на нас, а не наоборот

Разница здесь не в точке зрения — эти два варианта представляют разные ориентации трехмерного пространства, различимые с любой точки зрения.

Предположим, мы находимся на некоторой положительной координате на оси z, как фигурка на первом изображении на рис. 3.29. Мы должны видеть положительное направление оси y, повернутое на четверть оборота против часовой стрелки от положительного направления оси x, в противном случае оси будут ориентированы неправильно.

Многие вещи в реальном мире имеют ориентацию и не выглядят идентичными своим зеркальным отражениям. Например, левый и правый ботинки имеют одинаковые размер и форму, но разную ориентацию. Обычная кофейная кружка не имеет ориентации: мы не можем, глядя на две кофейные кружки без рисунков, сказать, различаются ли они. Но, как показано на рис. 3.30, две кофейные кружки с рисунками на противоположных сторонах можно различить.

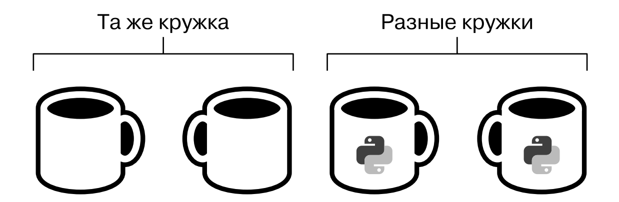


Рис. 3.30. Две кружки с рисунками на противоположных сторонах можно отличить друг от друга, чего не скажешь о кружках без рисунков

Самый доступный объект, который большинство математиков используют для определения ориентации, — это рука. Наши руки — это ориентированные объекты, поэтому мы можем отличить правую руку от левой, даже если она, к несчастью, была оторвана от тела. Можете ли вы сказать, какая рука изображена на рис. 3.31, правая или левая?

Очевидно, что это правая рука: ногти подсказывают, что мы видим тыльную сторону правой ладони! Математики используют свои руки для различения двух возможных ориентаций координатных осей и называют эти ориентации правосторонней и левосторонней. На рис. 3.32 показано, как работает это правило: если принять, что указательный палец правой руки направлен в положительном направлении вдоль оси x, то три других согнутых под прямым углом пальца будут направлены в положительном направлении вдоль оси y, а большой — вдоль оси z.

Это правило называется *правилом правой руки*, и если у вас оси соответствуют ему, то вы используете верную правостороннюю систему координат. Ориентация имеет значение! В программе управления дроном или роботом для лапароскопической хирургии все движения вверх, вниз, влево, вправо, вперед и назад должны быть согласованными. Векторное произведение — это ориентированная машина, поэтому оно может помочь следить за ориентацией во всех наших вычислениях.

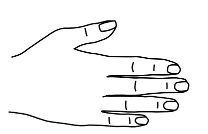


Рис. 3.31. Какая это рука, правая или левая?

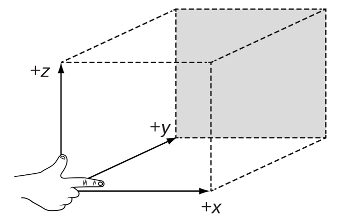


Рис. 3.32. Правило правой руки поможет вам вспомнить выбранную нами ориентацию осей системы координат

3.4.2. Определение направления с помощью векторного произведения

И снова, прежде чем продолжить рассказывать, как вычислить векторное произведение, я хочу показать, как оно выглядит. Пусть у нас есть два вектора, векторное произведение дает в результате вектор, перпендикулярный обоим. Например, если **u** = (1, 0, 0) и **v** = (0, 1, 0), то векторное произведение **u** × **v** даст вектор (0, 0, 1), как показано на рис. 3.33.

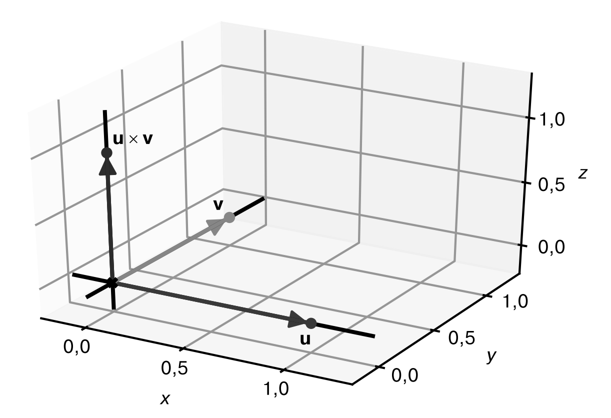


Рис. 3.33. Векторное произведение векторов **u** = (1, 0, 0) и **v** = (0, 1, 0)

На самом деле, как показано на рис. 3.34, векторное произведение любых двух векторов, лежащих на плоскости *xy*, лежит на оси *z*.

Отсюда становится ясно, почему векторное произведение невозможно вычислить в двух измерениях: оно возвращает вектор, лежащий за пределами плоскости, содержащей два исходных вектора. Результат перекрестного произведения перпендикулярен обоим входам, даже если они не лежат в плоскости *xy* (рис. 3.35).

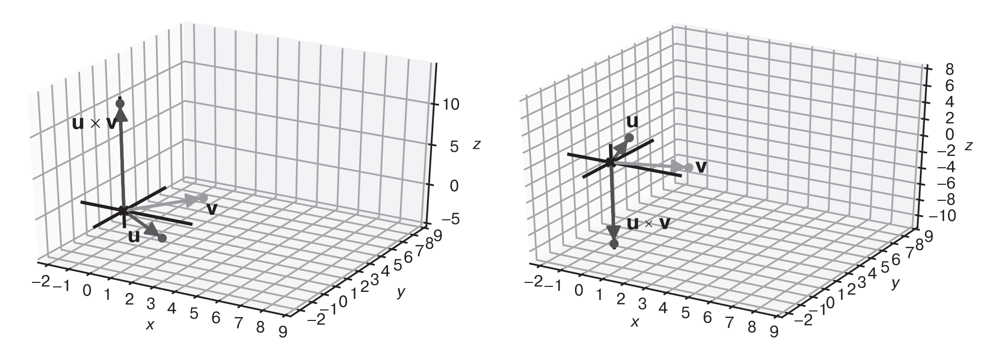


Рис. 3.34. Векторное произведение любых двух векторов, лежащих на плоскости *xy*, лежит на оси *z*

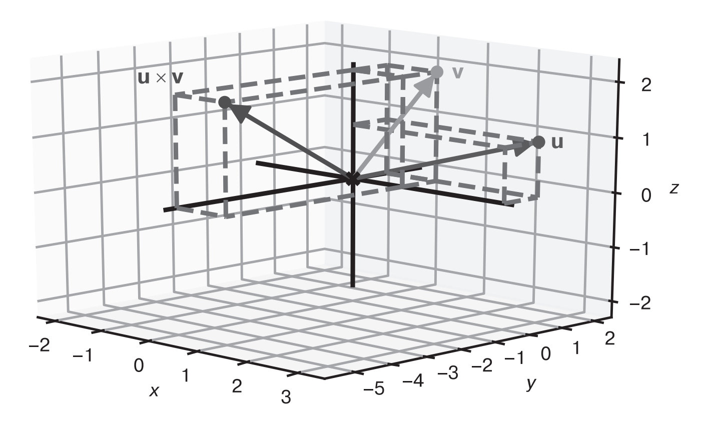


Рис. 3.35. Векторное произведение всегда возвращает вектор, перпендикулярный обоим исходным векторам

Но возможных перпендикулярных направлений два, а векторное произведение выбирает только одно. Например, произведение $(1, 0, 0) \times (0, 1, 0)$ дает в результате вектор $(0, 0, 1)$, указывающий в положительном направлении z. Любой вектор на оси z независимо от направления, в котором он указывает, будет перпендикулярен обоим исходным векторам. Но почему результат указывает в положительном направлении?

Вот тут-то и вступает в игру ориентация: векторное произведение подчиняется правилу правой руки. Векторное произведение $\mathbf{u} \times \mathbf{v}$ определяет направление, перпендикулярное исходным векторам $\mathbf{u}$ и $\mathbf{v}$, и переводит три вектора, $\mathbf{u}$, $\mathbf{v}$ и $\mathbf{u} \times \mathbf{v}$, в правостороннюю систему координат. То есть можно направить указательный палец правой руки в направлении $\mathbf{u}$, тогда три остальных согнутых пальца будут указывать в направлении $\mathbf{v}$, а большой — в направлении $\mathbf{u} \times \mathbf{v}$ (рис. 3.36).

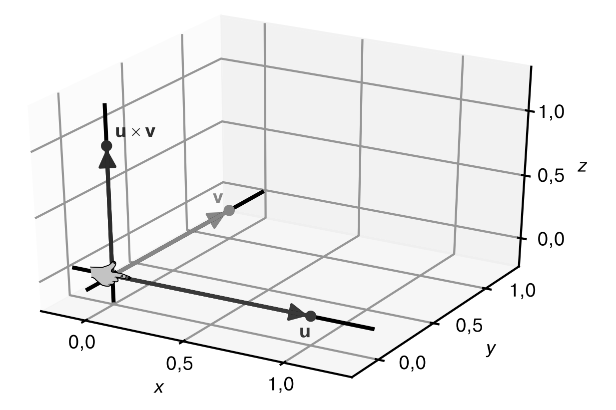


Рис. 3.36. Правило правой руки сообщает, *куда* направлен перпендикуляр векторного произведения

Когда исходные векторы лежат на двух осях координат, то несложно найти точное направление, в котором будет указывать их векторное произведение, — это одно из двух направлений вдоль оставшейся оси. В общем случае трудно определить направление, перпендикулярное двум векторам, без вычисления их векторного произведения. Это одна из особенностей, которая делает его таким полезным. Но вектор не просто указывает направление, он также определяет длину. Длина векторного произведения тоже несет полезную информацию.

3.4.3. Определение длины векторного произведения

Подобно скалярному произведению, длина векторного произведения — это число, дающее информацию об относительной ориентации исходных векторов, но вместо степени сонаправленности она сообщает величину, отражающую степень перпендикулярности, а если точнее, то площадь, охватываемую исходными векторами (рис. 3.37).

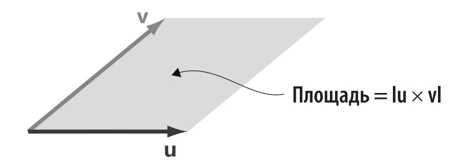


Рис. 3.37. Длина векторного произведения равна площади параллелограмма

Параллелограмм, ограниченный векторами **u** и **v**, как показано на рис. 3.37, имеет площадь, равную длине векторного произведения **u** × **v**. Любые два вектора заданной длины охватывают наибольшую площадь, если они перпендикулярны. В то же время, если **u** и **v** направлены в одну сторону, они не охватывают никакой области: длина векторного произведения равна нулю. Это удобное свойство, потому что не позволяет выбрать уникальное перпендикулярное направление, если исходные векторы параллельны.

В сочетании с направлением результат дает нам точный вектор. Векторное произведение двух векторов на плоскости гарантированно указывает в направлении $+z$ или $-z$. Как показано на рис. 3.38, чем больше площадь параллелограмма, образованного векторами на плоскости, тем больше векторное произведение.

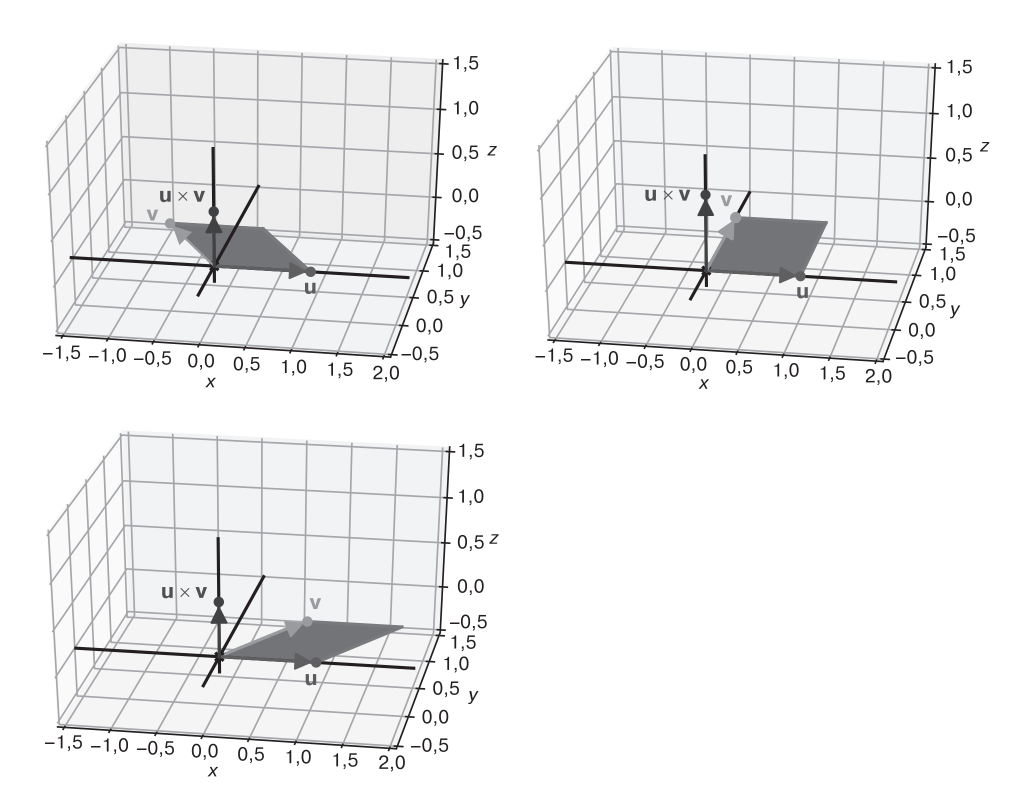


Рис. 3.38. Пары векторов на плоскости *xy* имеют разные значения векторных произведений в зависимости от площади охватываемого ими параллелограмма

Есть тригонометрическая формула площади этого параллелограмма: если **u** и **v** образуют угол θ, то площадь определяется как произведение $|\mathbf{u}| \cdot |\mathbf{v}| \cdot \sin\theta$. Рассмотрим некоторые простые векторные произведения с точки зрения их направлений и длин. Например, исследуем векторное произведение (0, 2, 0) и (0, 0, –2). Эти векторы лежат на осях y и z соответственно, поэтому векторное произведение согласно свойству перпендикулярности исходным векторам должно лежать на оси x. Найдем направление результата по правилу правой руки.

Повернув указательный палец в направлении первого вектора (положительное направление y) и согнув пальцы в направлении второго вектора (отрицательное направление z), мы обнаружим, что большой палец указывает в отрицательном направлении x. Векторное произведение равно $2 \cdot 2 \cdot \sin 90°$, потому что оси y

и z пересекаются под углом 90°. (В этом случае параллелограмм имеет форму квадрата со стороной 2.) Получается 4, то есть результат (−4, 0, 0) — это вектор с длиной 4, указывающий в направлении $-x$.

Геометрические вычисления подтверждают, что векторное произведение — это четко определенная операция. Но в общем случае исходные векторы не всегда лежат на осях и не очевидно, какие координаты использовать, чтобы найти перпендикулярный результат. К счастью, существует явная формула вычисления координат векторного произведения через координаты исходных векторов.

3.4.4. Вычисление векторного произведения трехмерных векторов

На первый взгляд формула векторного произведения выглядит запутанной, но ее легко можно превратить в функцию на Python, чтобы применять в будущих вычислениях. Начнем с координат двух векторов, **u** и **v**. Мы могли бы обозначить координаты как $\mathbf{u} = (a, b, c)$ и $\mathbf{v} = (d, e, f)$, но будет понятнее, если использовать более удобные обозначения $\mathbf{u} = (u_x, u_y, u_z)$ и $\mathbf{v} = (v_x, v_y, v_z)$. Обозначение v_x явно показывает, что это координата x вектора **v**, чего не скажешь о простом однобуквенном обозначении, например d. В терминах этих координат формула векторного произведения выглядит так:

$$\mathbf{u} \times \mathbf{v} = (u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x)$$

или на Python:

```
def cross(u, v):
    ux,uy,uz = u
    vx,vy,vz = v
    return (uy*vz - uz*vy, uz*vx - ux*vz, ux*vy - uy*vx)
```

Вы сможете протестировать эту формулу в упражнениях. Обратите внимание на то, что в отличие от большинства формул, которые мы использовали до сих пор, эта довольно плохо обобщается на другое число измерений. Она требует, чтобы исходные векторы были трехмерными.

Эта алгебраическая процедура согласуется с геометрическим описанием в текущей главе. Векторное произведение сообщает нам площадь и направление, соответственно, оно помогает решить, увидит ли обитатель трехмерного пространства многоугольник, парящий в пространстве рядом с ним. Например, как показано на рис. 3.39, наблюдатель, стоящий на оси x, не увидит параллелограмм, образованный векторами $\mathbf{u} = (1, 1, 0)$ и $\mathbf{v} = (-2, 1, 0)$. Другими словами, многоугольник на рис. 3.39 параллелен линии взгляда наблюдателя. Используя векторное произведение, мы могли бы сказать это, не рисуя картинку. Если векторное произведение перпендикулярно линии взгляда наблюдателя, то параллелограмм не будет ему виден.

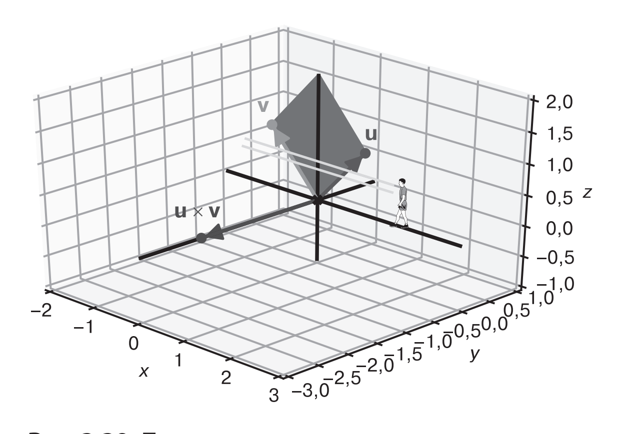
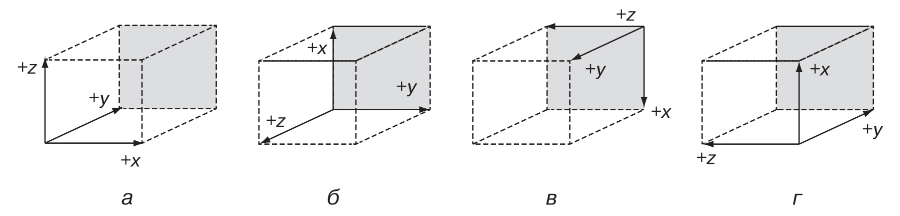


Рис. 3.39. Применяя векторное произведение, можно сказать, будет ли виден параллелограмм наблюдателю

Теперь пришло время для кульминационного проекта — создания трехмерного объекта из многоугольников и его отображения на двухмерном холсте. Для этого мы используем все векторные операции, рассмотренные до сих пор. В частности, векторное произведение поможет нам решить, какие многоугольники видны.

3.4.5. Упражнения

Упражнение 3.19. На каждой из следующих схем показаны три взаимно перпендикулярные стрелки, указывающие в положительных направлениях *x*, *y* и *z*. Для придания визуальной глубины задняя грань трехмерного блока окрашена в серый цвет. Какая из четырех схем совместима с выбранной нами ориентацией системы координат? То есть на какой из них оси *x*, *y* и *z* располагаются так, как мы их нарисовали, пусть даже с другой точки зрения?



Какая из этих схем совместима с выбранной нами ориентацией системы координат?

Решение. Если взглянуть на схему *а* сверху, то мы увидим, что оси *x* и *y* направлены как обычно и ось *z* указывает на нас. Схема *а* соответствует выбранной нами ориентации системы координат.

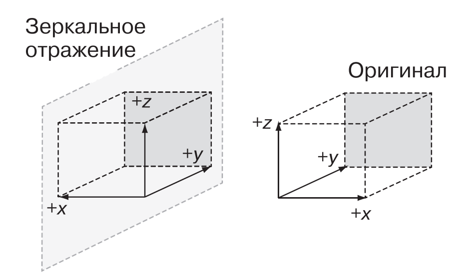
На схеме *б* ось z указывает на нас, но ось $+y$ повернута относительно оси $+x$ на 90° по часовой стрелке. Это не соответствует выбранной ориентации.

Если посмотреть на схему *в* из точки, куда направлен положительный конец оси z (с левой стороны прямоугольника), то мы увидим, что ось $+y$ повернута относительно оси $+x$ на 90° против часовой стрелки. Схема *с* соответствует выбранной нами ориентации.

Если посмотреть на схему *г* слева, куда направлен положительный конец оси $+z$, то мы снова увидим, что ось $+y$ повернута относительно оси $+x$ на 90° против часовой стрелки. Эта схема тоже соответствует выбранной нами ориентации.

Упражнение 3.20. Если поместить три оси координат перед зеркалом, будет ли отражение в зеркале иметь ту же ориентацию?

Решение. Зеркальное отражение имеет обратную ориентацию. С этой точки зрения оси z и y остаются направленными в том же направлении. В оригинальной системе координат ось x повернута относительно оси y по часовой стрелке, а в зеркальном отражении — против часовой стрелки.



Оси *x*, *y* и *z* и их зеркальное отражение

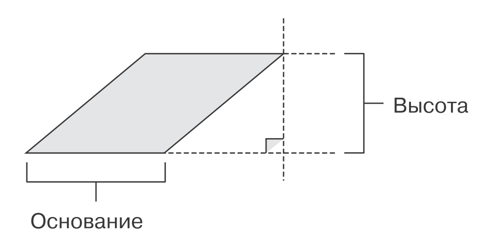
Упражнение 3.21. В каком направлении указывает результат векторного произведения $(0, 0, 3) \times (0, -2, 0)$?

Решение. Если сориентировать указательный палец правой руки в направлении $(0, 0, 3)$, то есть положительном направлении оси z, а остальные пальцы согнуть и направить на $(0, -2, 0)$, то есть в отрицательном направлении оси y, то большой палец будет соответствовать положительному направлению оси x. Следовательно, $(0, 0, 3) \times (0, -2, 0)$ указывает в положительном направлении x.

Упражнение 3.22. Вычислите координаты векторного произведения векторов (1, –2, 1) и (–6, 12, –6).

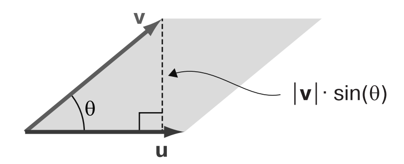
Решение. Так как координаты этих векторов кратны и противоположны по знаку, они указывают в противоположных направлениях и не охватывают никакой области, поэтому длина векторного произведения равна нулю. Единственный вектор с нулевой длиной — это (0, 0, 0), так что это и есть ответ.

Упражнение 3.23. Мини-проект. Площадь параллелограмма равна произведению длины его основания на высоту, как показано здесь.



Учитывая это, объясните, почему для расчета площади подходит формула $|\mathbf{u}| \cdot |\mathbf{v}| \cdot \sin\theta$.

Решение. На схеме вектор **u** определяет основание, поэтому длина основания равна $|\mathbf{u}|$. Высота, проведенная от конца вектора **v** до основания, образует прямоугольный треугольник. Длина **v** — это гипотенуза, а вертикальный катет треугольника — это высота, которую мы ищем. По определению функции синуса высота равна $|\mathbf{v}| \cdot \sin\theta$.



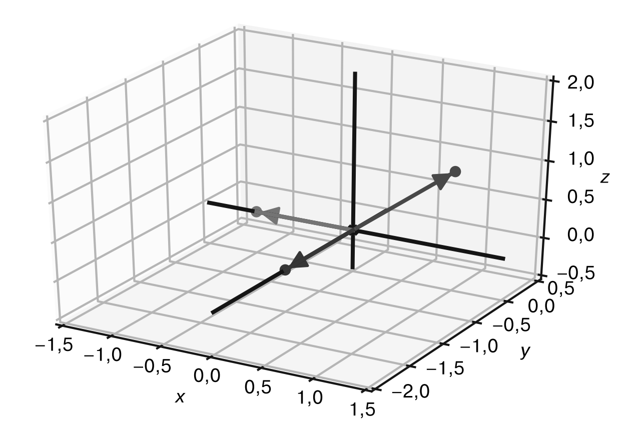
Формула площади параллелограмма в терминах синуса одного из углов

Поскольку длина основания равна $|\mathbf{u}|$, а высота равна $|\mathbf{v}| \cdot \sin\theta$, площадь параллелограмма действительно вычисляется по формуле $|\mathbf{u}| \cdot |\mathbf{v}| \cdot \sin\theta$.

Упражнение 3.24. Вычислите результат векторного произведения $(1, 0, 1) \times (-1, 0, 0)$. Варианты ответа:

1. $(0, 1, 0)$.
2. $(0, -1, 0)$.
3. $(0, -1, -1)$.
4. $(0, 1, -1)$.

Решение. Эти векторы лежат на плоскости xz, поэтому их векторное произведение лежит на оси y. Если указательным пальцем правой руки указать в направлении $(1, 0, 1)$ и согнуть пальцы в направлении $(-1, 0, 0)$, то большой палец будет указывать в направлении $-y$.



Вычисление векторного произведения (1, 0, 1) и (–1, 0, 0) геометрическим способом

Мы могли бы найти длины векторов и угол между ними, чтобы вычислить векторное произведение, но у нас уже есть основание и высота, равные 1. Поэтому длина равна 1, а векторное произведение — это вектор $(0, -1, 0)$ с длиной 1, направленный в сторону $-y$, то есть верный ответ — 2.

Упражнение 3.25. Используя функцию `cross`, вычислите $(0, 0, 1) \times \mathbf{v}$ для нескольких различных значений вектора $\mathbf{v}$. Чему равна z каждого результата и почему?

Решение. Независимо от выбора вектора **v** координата z результата равна нулю:

```
>>> cross((0,0,1),(1,2,3))
(-2, 1, 0)
>>> cross((0,0,1),(-1,-1,0))
(1, -1, 0)
>>> cross((0,0,1),(1,-1,5))
(1, 1, 0)
```

Координаты u_x и u_y вектора $\mathbf{u} = (0, 0, 1)$ равны нулю, поэтому член $u_xv_y - u_yv_x$ в формуле векторного произведения равен нулю независимо от значений v_x и v_y. Это имеет следующий геометрический смысл: векторное произведение должно быть перпендикулярно обоим векторам-сомножителям, а чтобы быть перпендикулярным вектору (0, 0, 1), компонента z результата должна быть равна нулю.

Упражнение 3.26. Мини-проект. Покажите алгебраически, что векторное произведение $\mathbf{u} \times \mathbf{v}$ перпендикулярно обоим векторам, **u** и **v**, независимо от их координат.

Подсказка. Покажите вычисление $(\mathbf{u} \times \mathbf{v}) \cdot \mathbf{u}$ и $(\mathbf{u} \times \mathbf{v}) \cdot \mathbf{v}$, разложив векторы на координаты.

Решение. Пусть $\mathbf{u} = (u_x, u_y, u_z)$ и $\mathbf{v} = (v_x, v_y, v_z)$. Мы можем записать $(\mathbf{u} \times \mathbf{v}) \cdot \mathbf{u}$ в терминах координат следующим образом:

$$(\mathbf{u} \times \mathbf{v}) \cdot \mathbf{u} = (u_yv_z - u_zv_y, u_zv_x - u_xv_z, u_xv_y - u_yv_x) \cdot (u_x, u_y, u_z).$$

Скалярное умножение векторного произведения на вектор

Раскрыв скобки, получаем шесть членов, компенсирующих друг друга:

$$(u_yv_z - u_zv_y)u_x + (u_zv_x - u_xv_z)u_y + (u_xv_y - u_yv_x)u_z =$$
$$= u_yv_zu_x - u_zv_yu_x + u_zv_xu_y - u_xv_zu_y + u_xv_yu_z - u_yv_xu_z.$$

После сокращения в правой части уравнения ничего не остается

Поскольку все члены сократились, результат равен нулю. Чтобы сэкономить «чернила», я не буду показывать результат $(\mathbf{u} \times \mathbf{v}) \cdot \mathbf{v}$, но в этом случае происходит то же самое: появляются шесть членов, которые взаимно сокращаются, в итоге получается ноль. Это означает, что векторное произведение $(\mathbf{u} \times \mathbf{v})$ перпендикулярно обоим векторам, **u** и **v**.

3.5. ОТОБРАЖЕНИЕ ТРЕХМЕРНОГО ОБЪЕКТА НА ДВУХМЕРНОЙ ПЛОСКОСТИ

Попробуем использовать все, что мы узнали, для визуализации простой трехмерной фигуры, которая называется октаэдром. В отличие от куба с шестью квадратными гранями, октаэдр имеет восемь треугольных граней. Октаэдр можно представить как две четырехгранные пирамиды с общим основанием. На рис. 3.40 показан скелет октаэдра.

Если бы это было непрозрачное тело, противоположные стороны были бы невидимыми и мы видели бы только четыре грани из восьми, как показано на рис. 3.41.

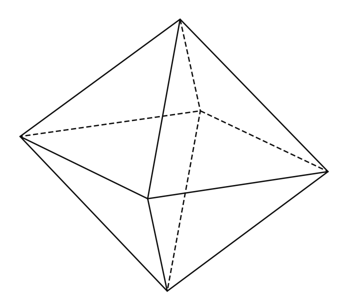


Рис. 3.40. Скелет октаэдра — фигуры с восемью гранями и шестью вершинами. Пунктирными линиями показаны ребра октаэдра на противоположной от нас стороне

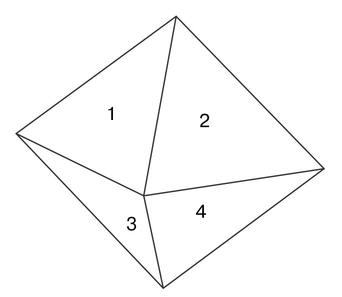


Рис. 3.41. Четыре пронумерованные грани, видимые в текущем положении октаэдра

Отображение октаэдра сводится к определению четырех треугольников, которые нужно показать, и степени их затенения. Посмотрим, как это сделать.

3.5.1. Определение трехмерного объекта с помощью векторов

Октаэдр — простая фигура, потому что имеет всего шесть вершин. Мы можем задать для них простые координаты (1, 0, 0), (0, 1, 0), (0, 0, 1) и три противоположных вектора, как показано на рис. 3.42.

Эти шесть векторов определяют границы фигуры, но не дают всей информации, необходимой для ее рисования. Нам также нужно определить, какие из этих вершин должны соединяться отрезками, чтобы сформировать ребра. Например,

верхняя точка на рис. 3.42, имеющая координаты (0, 0, 1), соединяется ребром со всеми четырьмя точками на плоскости xy (рис. 3.43).

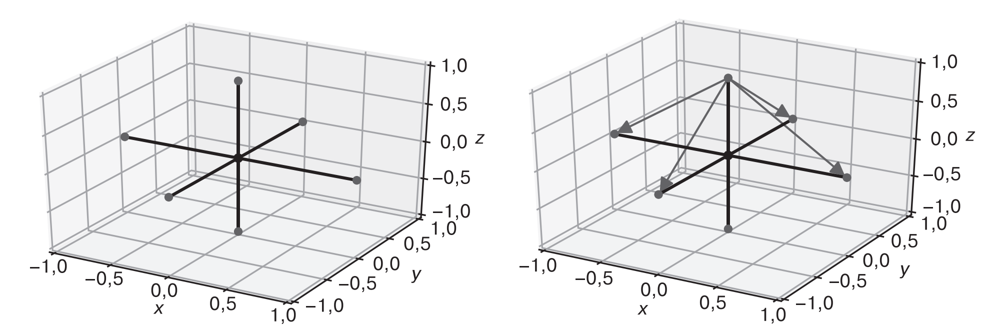


Рис. 3.42. Вершины октаэдра

Рис. 3.43. Четыре ребра октаэдра, обозначенные стрелками

Эти ребра очерчивают верхнюю пирамиду октаэдра. Обратите внимание на то, что на рисунке отсутствуют ребра от (0, 0, 1) до (0, 0, –1), потому что этот сегмент будет находиться внутри октаэдра. Каждое ребро определяется парой векторов: начальной и конечной точками отрезка ребра. Например, (0, 0, 1) и (1, 0, 0) определяют одно из ребер.

Но и ребер недостаточно для завершения рисунка. Нам также нужно знать, какие тройки вершин и ребер определяют треугольные грани, которые мы будем заливать цветом. Вот тут-то и пригодится понимание ориентации: нам нужно знать не только, какие сегменты определяют грани октаэдра, но также обращены ли они к нам или от нас.

Далее мы используем следующую стратегию: определим треугольную лицевую поверхность грани как три вектора $\mathbf{v}_1$, $\mathbf{v}_2$ и $\mathbf{v}_3$, задающие ребра. (Обратите внимание: здесь индексы 1, 2 и 3 применяются для различения трех разных векторов, а не компонент одного и того же вектора.) В частности, мы упорядочим $\mathbf{v}_1$, $\mathbf{v}_2$ и $\mathbf{v}_3$ так, что $(\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{v}_3 - \mathbf{v}_1)$ будет направлен наружу из октаэдра (рис. 3.44). Если вектор, направленный наружу, указывает в нашу сторону, это означает, что с нашей точки обзора лицевая сторона грани видна. Иначе она не видна и ее не нужно рисовать.

Восемь треугольных граней можно определить как тройки векторов $\mathbf{v}_1$, $\mathbf{v}_2$ и $\mathbf{v}_3$:

```
octahedron = [
    [(1,0,0), (0,1,0), (0,0,1)],
    [(1,0,0), (0,0,-1), (0,1,0)],
    [(1,0,0), (0,0,1), (0,-1,0)],
    [(1,0,0), (0,-1,0), (0,0,-1)],
    [(-1,0,0), (0,0,1), (0,1,0)],
    [(-1,0,0), (0,1,0), (0,0,-1)],
    [(-1,0,0), (0,-1,0), (0,0,1)],
    [(-1,0,0), (0,0,-1), (0,-1,0)],
]
```

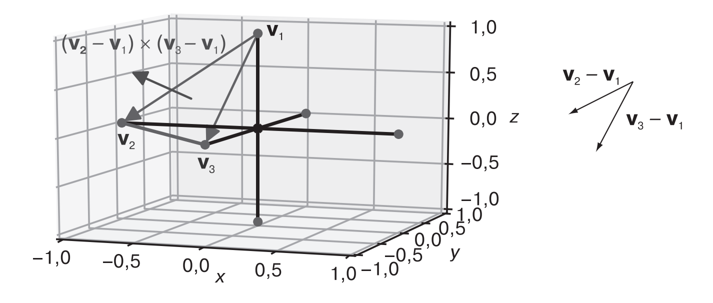


Рис. 3.44. Лицевая сторона грани октаэдра. Три точки определяют лицевую сторону грани так, что вектор $(\mathbf{v}_2 \cdot \mathbf{v}_1) \times (\mathbf{v}_3 \cdot \mathbf{v}_1)$ направлен наружу из фигуры

На самом деле лицевые стороны — это единственные данные, которые нужны для визуализации фигуры, — они неявно содержат ребра и вершины. Вершины граней, например, можно получить с помощью следующей функции:

```
def vertices(faces):
    return list(set([vertex for face in faces for vertex in face]))
```

3.5.2. Проецирование на двухмерную плоскость

Чтобы спроецировать на двухмерную плоскость трехмерную фигуру, мы должны выбрать направление в трехмерном пространстве, с которого будем смотреть на нее. Имея два трехмерных вектора, определяющих направления «вверх» и «вправо» с нашей точки обзора, можно *спроецировать* на них любой трехмерный вектор и получить две компоненты вместо трех. Функция `component` извлекает часть любого трехмерного вектора, указывающую в заданном направлении, используя скалярное произведение:

```
def component(v,direction):
    return (dot(v,direction) / length(direction))
```

С двумя жестко заданными направлениями, в данном случае $(1, 0, 0)$ и $(0, 1, 0)$, можно определить способ проецирования трехмерного вектора на двухмерную плоскость. Следующая функция принимает трехмерный вектор, или кортеж из трех чисел, и возвращает двухмерный, или кортеж из двух чисел:

```
def vector_to_2d(v):
    return (component(v,(1,0,0)), component(v,(0,1,0)))
```

Мы можем изобразить полученный вектор как проекцию трехмерного вектора на плоскость. Удаление компоненты z лишает вектор глубины (рис. 3.45).

Наконец, чтобы получить проекцию трехмерного треугольника на плоскость, нужно применить эту функцию ко всем вершинам, определяющим грань:

```
def face_to_2d(face):
    return [vector_to_2d(vertex) for vertex in face]
```

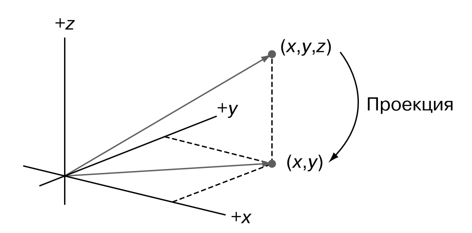


Рис. 3.45. Удаление компоненты *z* из трехмерного вектора превращает его в плоскую проекцию на плоскости *xy*

3.5.3. Ориентация лицевой стороны и затенение

Чтобы изобразить тени на двухмерном рисунке, нужно выбрать фиксированный цвет для каждого треугольника в зависимости от угла падения света на него. Допустим, источник света находится в точке (1, 2, 3). Тогда яркость освещенности лицевой стороны треугольной грани будет зависеть от величины угла между перпендикуляром, проведенным к грани, и направлением на источник света — чем меньше угол, тем ярче освещается грань. Нам не нужно беспокоиться о вычислении цветов — в Matplotlib есть встроенная библиотека, которая сделает это сама. Например, вызов

```
blues = matplotlib.cm.get_cmap('Blues')
```

вернет функцию, ссылку на которую мы сохраняем в `blues`, отображающую числа от 0 до 1 в спектр значений от темно- до ярко-синего. Наша задача — найти число от 0 до 1, определяющее яркость освещенности лицевой стороны грани.

Имея вектор, перпендикулярный к грани (или *нормаль*), и вектор, направленный на источник света, можно вычислить их скалярное произведение и получить степень их сонаправленности. Кроме того, поскольку нас интересуют только направления, мы можем использовать векторы длиной 1. Тогда, если лицевая грань хоть в какой-то степени обращена к источнику света, скалярное произведение даст нам величину между 0 и 1. Если угол между нормалью и направлением на источник света больше 90°, то такая грань вообще не освещается и не должна отображаться. Следующая вспомогательная функция принимает вектор и возвращает другой вектор того же направления, но длиной 1:

```
def unit(v):
    return scale(1./length(v), v)
```

Другая вспомогательная функция принимает грань и возвращает перпендикулярный к ней вектор:

```
def normal(face):
    return(cross(subtract(face[1], face[0]), subtract(face[2], face[0])))
```

Объединив все вместе, получаем функцию, которая рисует все треугольники, составляющие трехмерную фигуру, используя функцию `draw` (я переименовал

`draw` в `draw2d` и соответственно переименовал классы, чтобы отличить их от трехмерных аналогов):

```
def render(faces, light=(1,2,3), color_map=blues, lines=None):
    polygons = []
    for face in faces:
        unit_normal = unit(normal(face))
        if unit_normal[2] > 0:
            c = color_map(1 - dot(unit(normal(face)),
                                  unit(light)))
            p = Polygon2D(*face_to_2d(face),
                                  fill=c, color=lines)
            polygons.append(p)
    draw2d(*polygons,axes=False, origin=False, grid=None)
```

Для лицевой стороны каждой грани вычислить вектор нормали длиной 1

Продолжать вычисления, только если координата z этого вектора положительная, то есть если грань обращена лицевой стороной к наблюдателю

Чем больше скалярное произведение между вектором нормали и направлением на источник света, тем ярче освещена грань

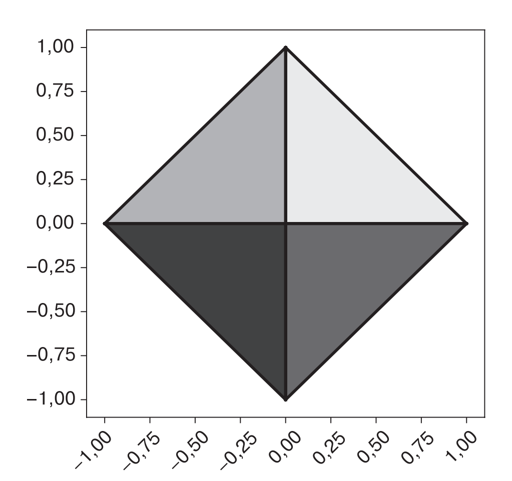
Необязательный аргумент для рисования линий по краям каждого треугольника, чтобы показать скелет фигуры

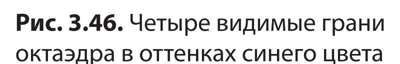
Чтобы реализовать функцию `render`, отображающую октаэдр, потребовалось всего несколько строк кода:

```
render(octahedron, color_map=matplotlib.cm.get_cmap('Blues'), lines=black)
```

На рис. 3.46 показан результат ее работы.

Октаэдр, нарисованный в оттенках синего (в черно-белой книге — в оттенках серого. — *Примеч. ред.*), не выглядит таким уж особенным, но если добавить больше граней, становится видно, что прием затенения работает (рис. 3.47). В примерах исходного кода к этой книге вы найдете готовые фигуры с большим количеством граней.




Рис. 3.46. Четыре видимые грани октаэдра в оттенках синего цвета

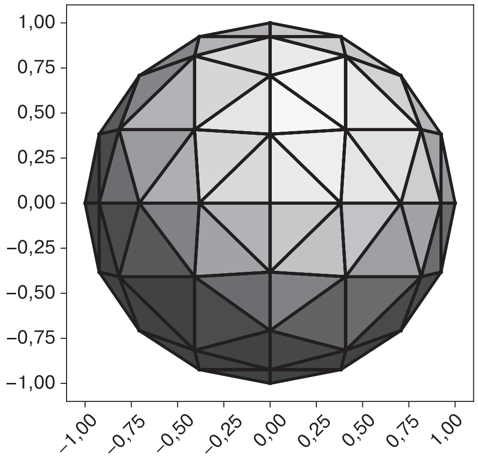


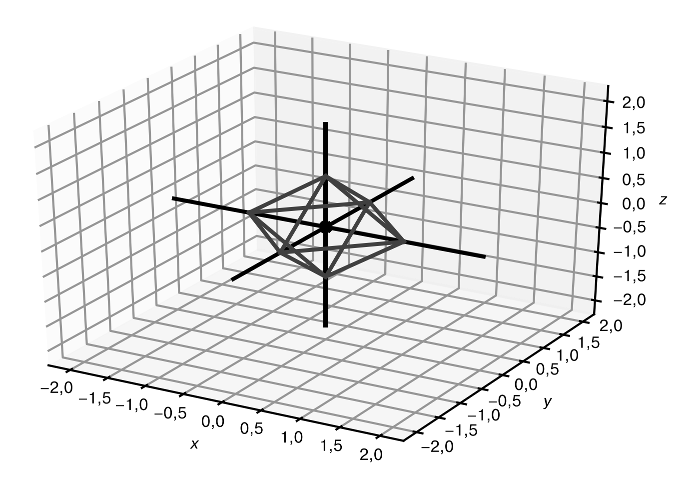
Рис. 3.47. Трехмерная фигура с большим количеством треугольных граней. Эффект затенения на примере этой фигуры более очевиден

3.5.4. Упражнения

Упражнение 3.27. Мини-проект. Найдите пары векторов, определяющие каждое из 12 ребер октаэдра, и нарисуйте все ребра с помощью кода на Python.

Решение. Вершина октаэдра $(0, 0, 1)$. Она соединяется ребрами с четырьмя точками на плоскости xy. Точно так же основание октаэдра $(0, 0, -1)$ соединяется ребрами с четырьмя точками на плоскости xy. Наконец, четыре точки на плоскости xy соединяются друг с другом, образуя квадрат:

```
top = (0,0,1)
bottom = (0,0,-1)
xy_plane = [(1,0,0),(0,1,0),(-1,0,0),(0,-1,0)]
edges = [Segment3D(top,p) for p in xy_plane] +\
            [Segment3D(bottom, p) for p in xy_plane] +\
            [Segment3D(xy_plane[i],xy_plane[(i+1)%4]) for i in
range(0,4)]
draw3d(*edges)
```



Ребра октаэдра

Упражнение 3.28. Первая грань октаэдра задается точками $[(1, 0, 0), (0, 1, 0), (0, 0, 1)]$. Это единственный допустимый порядок перечисления вершин для этой грани?

Решение. Не единственный. Например, $[(0, 1, 0), (0, 0, 1), (1, 0, 0)]$ — тот же набор из трех точек, и результат векторного произведения по-прежнему указывает в том же направлении.

КРАТКИЕ ИТОГИ ГЛАВЫ

- В отличие от двухмерного пространства, имеющего длину и ширину, трехмерное пространство имеет еще и глубину.
- Трехмерные векторы определяются тремя координатами: x, y и z. Они сообщают, сколько шагов нужно сделать вдоль каждой оси от начала координат, чтобы добраться до трехмерной точки.
- Подобно двухмерным векторам, трехмерные векторы тоже можно складывать, вычитать и умножать на скаляры. Аналогично можно вычислять их длины, используя трехмерную аналогию теоремы Пифагора.
- Скалярное произведение — это операция умножения двух векторов, дающая в результате скаляр, который может служить оценкой сонаправленности двух векторов и применяться для вычисления угла между ними.
- Векторное произведение — это операция умножения двух векторов, дающая в результате третий вектор, перпендикулярный исходным. Результат векторного произведения — это площадь параллелограмма, охватываемого двумя исходными векторами.
- Поверхность любого трехмерного объекта можно представить в виде набора треугольников, каждый из которых определяется тремя векторами, обозначающими вершины.
- Используя векторное произведение, можно определить, виден ли тот или иной треугольник с некоторой точки в трехмерном пространстве и насколько он освещен источником света, находящимся в заданном местоположении. Нарисовав и затенив все треугольники, определяющие поверхность объекта, можно придать ему объемность.

4 Преобразование векторов и графики

В этой главе

- ✓ Преобразование и рисование трехмерных объектов с применением математических функций.
- ✓ Создание компьютерной анимации с использованием преобразований векторной графики.
- ✓ Идентификация линейных преобразований, сохраняющих линии и многоугольники.
- ✓ Вычисление влияния линейных преобразований на векторы и трехмерные модели.

Применив приемы из двух предыдущих глав и подключив смекалку, можно визуализировать любую двухмерную или трехмерную фигуру. Из отрезков линий и многоугольников, определяемых векторами, можно конструировать объекты, персонажи и даже целые миры. Но между вами и вашим первым полнометражным компьютерным анимационным фильмом или реалистичной игрой есть одно препятствие — вам нужно научиться рисовать объекты, которые *меняются* во времени.

Анимация компьютерной графики реализуется точно так же, как мультипликационные фильмы: создаются статические изображения, которые затем отображаются со скоростью несколько десятков кадров в секунду. Когда человек видит

так быстро сменяющиеся картинки, ему кажется, что изображение постоянно меняется. В главах 2 и 3 мы рассмотрели несколько математических операций, которые принимают существующие векторы и преобразуют их в новые. Соединив последовательность небольших преобразований в цепочку, можно создать иллюзию непрерывного движения.

Чтобы представить, как это делается, вспомните примеры поворота двухмерных векторов. Мы видели функцию `rotate` на Python, которая принимает двухмерный вектор и поворачивает его, скажем, на 45° против часовой стрелки. Как показано на рис. 4.1, функцию `rotate` можно рассматривать как машину, которая принимает исходный вектор и выдает преобразованный вектор.

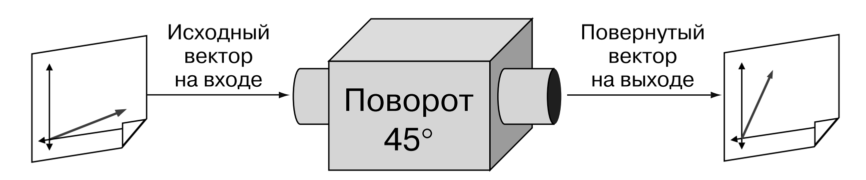


Рис. 4.1. Представление векторной функции в виде машины, поворачивающей вектор

Если применить трехмерный аналог этой функции к каждому вектору каждого многоугольника, определяющего трехмерную фигуру, то мы увидим, как поворачивается вся фигура, которая может быть октаэдром из предыдущей главы или иметь более интересную форму, например чайника. На рис. 4.2 показано, как машина, выполняющая поворот, принимает чайник на входе и выдает повернутую его копию на выходе.

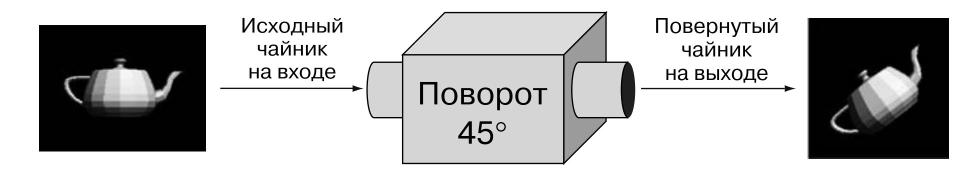


Рис. 4.2. Преобразование можно применить к каждому вектору, составляющему трехмерную модель, и тем самым преобразовать ее целиком

Если вместо одного поворота на 45° повернуть фигуру на один градус 45 раз, то можно сгенерировать кадры фильма, показывающего вращающийся чайник (рис. 4.3). Операция поворота — отличный пример, потому что при повороте каждой точки отрезка на один и тот же угол относительно начала координат получается отрезок той же длины. В результате, после поворота всех векторов, очерчивающих двух- или трехмерный объект, мы все равно можем узнать его.

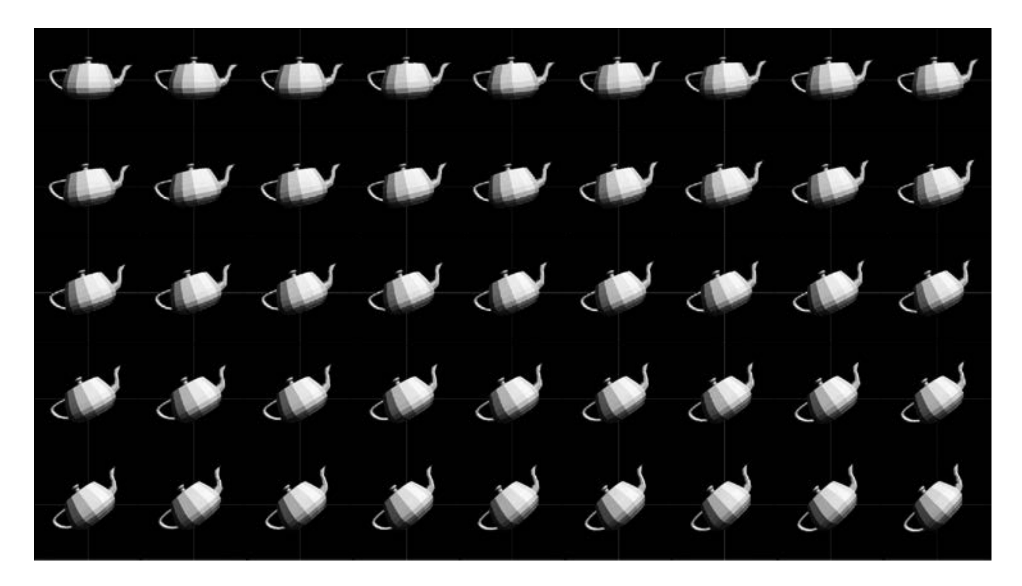


Рис. 4.3. Кадры, описывающие поворот чайника 45 раз на 1°

Я познакомлю вас с широким классом векторных преобразований, называемых *линейными преобразованиями*, которые, как и поворот, преобразуют векторы, лежащие на прямой, в новые векторы, также лежащие на прямой. Линейные преобразования широко применяются в математике, физике и анализе данных. Поэтому вам будет полезно знать, как изобразить их геометрически, встретившись с ними в этих контекстах.

Для визуализации поворотов, линейных и других векторных преобразований в этой главе мы начнем использовать более мощные инструменты рисования. Matplotlib заменим на OpenGL — стандартную высокопроизводительную библиотеку отображения графики. В большинстве случаев для работы с OpenGL применяется язык программирования C или C++, но мы будем работать с ней из Python с помощью удобной обертки PyOpenGL. Задействуем также библиотеку PyGame, предназначенную для разработки видеоигр на Python. В частности, применим функции из PyGame, упрощающие преобразование последовательных изображений в анимацию. Настройка всех этих новых инструментов описана в приложении B, поэтому сразу перейдем к их использованию и сосредоточимся на математике преобразования векторов. Если вы собираетесь опробовать примеры кода из этой главы (что я настоятельно рекомендую!), то прямо сейчас перейдите к приложению B и возвращайтесь, когда установите и настроите все необходимое.

4.1. ПРЕОБРАЗОВАНИЕ ТРЕХМЕРНЫХ ОБЪЕКТОВ

Наша главная цель в этой главе — взять трехмерный объект, например чайник, и изменить его, чтобы создать новый трехмерный объект, визуально отличающийся от исходного. В главе 2 мы уже видели, что можно выполнить параллельный перенос или масштабирование каждого вектора, из которых состоит двухмерный динозавр, и вся его фигура будет двигаться или масштабироваться соответственно. Здесь применим тот же подход. Каждое рассматриваемое нами преобразование принимает и возвращает вектор, как показано в следующем псевдокоде:

```
def transform(v):
    old_x, old_y, old_z = v
    # ... выполнить здесь некоторые вычисления ...
    return (new_x, new_y, new_z)
```

Начнем с уже знакомых параллельного переноса и масштабирования.

4.1.1. Рисование преобразованного объекта

Если вы уже установили все необходимое, как описано в приложении В, то сможете запустить файл `draw_teapot.py`, находящийся в папке с примерами для главы 4 (инструкции по запуску сценариев на Python из командной строки найдете в приложении А). В случае успеха вы должны увидеть окно PyGame с изображением, показанным на рис. 4.4.



Рис. 4.4. Результат выполнения сценария draw_teapot.py

В следующих нескольких примерах мы будем применять преобразования к векторам, определяющим чайник, и снова отображать его, чтобы увидеть произведенный эффект. В качестве первого примера масштабируем все векторы на одну и ту же величину. Следующая функция `scale2` умножает входной вектор на скаляр 2,0 и возвращает результат:

```
from vectors import scale
def scale2(v):
    return scale(2.0, v)
```

Функция `scale2(v)` имеет ту же организацию, что и функция `transform(v)`, приведенная в начале этого раздела: она получает исходный трехмерный вектор и возвращает новый трехмерный вектор. Чтобы применить это преобразование к чайнику, нужно преобразовать каждую его вершину. Это можно сделать, перебирая треугольник за треугольником. Для каждого треугольника, составляющего исходный чайник, мы создадим новый треугольник, используя результаты применения `scale2` к исходным вершинам:

```
original_triangles = load_triangles()   ← Загрузить треугольники, используя код из приложения В
scaled_triangles = [
    [scale2(vertex) for vertex in triangle]   ← Применить scale2 к каждой вершине данного треугольника, чтобы получить новые вершины
    for triangle in original_triangles   ← Повторить для каждого треугольника в списке
]
```

Теперь, получив новый набор треугольников, мы можем нарисовать их, вызвав функцию `draw_model(scaled_triangles)`. На рис. 4.5 показан чайник после этого вызова, и вы можете воспроизвести его, запустив сценарий `scale_teapot.py`.

Этот чайник выглядит крупнее оригинала. На самом деле он вдвое больше, потому что мы умножили каждый вектор на 2. Применим к каждому вектору еще одно преобразование — параллельный перенос на вектор (−1, 0, 0).

Напомню, что перенос на вектор — это другое название операции сложения векторов, поэтому на самом деле я подразумеваю прибавление вектора (−1, 0, 0) к каждой вершине, образующей чайник. В результате этого он должен переместиться на одну единицу в отрицательном направлении x, то есть с нашей точки зрения влево. Перенос одной вершины выполняет следующая функция:

```
from vectors import add
def translate1left(v):
    return add((-1,0,0), v)
```

Теперь, начав с исходного набора треугольников, масштабируем все вершины, а затем применим к ним операцию параллельного переноса. Результат показан

на рис. 4.6. Вы можете воспроизвести его у себя, запустив сценарий `scale_translate_teapot.py`:

```
scaled_translated_triangles = [
    [translate1left(scale2(vertex)) for vertex in triangle]
    for triangle in original_triangles
]
draw_model(scaled_translated_triangles)
```

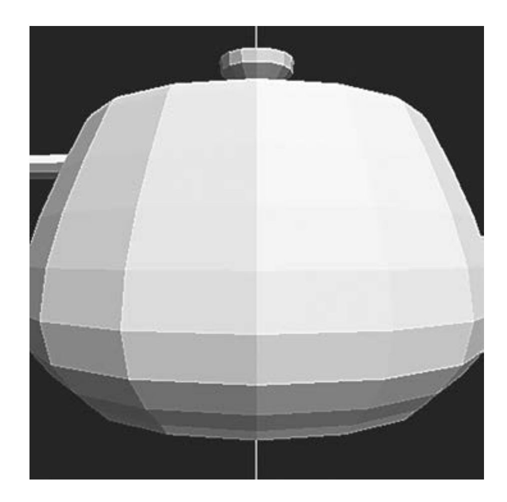


Рис. 4.5. Применение scale2 к каждой вершине каждого треугольника дает чайник с размерами в два раза больше первоначальных



Рис. 4.6. Увеличенный и смещенный чайник

Разные скалярные множители по-разному изменяют размер чайника, и разные векторы параллельного переноса перемещают его в разные места в пространстве. В упражнениях в конце этого раздела у вас будет возможность попробовать разные скалярные множители и векторы переноса, а сейчас сосредоточимся на комбинировании и применении большего количества преобразований.

4.1.2. Комбинирование векторных преобразований

Последовательное применение любого количества преобразований определяет новое преобразование. В предыдущем разделе, например, мы масштабировали и затем переместили чайник. Можно упаковать эту последовательность преобразований в новое преобразование, оформив в виде отдельной функции на Python:

```
def scale2_then_translate1left(v):
    return translate1left(scale2(v))
```

Это важный принцип! Векторные преобразования принимают и возвращают векторы, поэтому их можно комбинировать как угодно и сколько угодно, используя прием *композиции функций*. Для тех, кто прежде не знал этого термина, отмечу, что он означает определение новых функций, которые применяют две или более существующие функции в некоем порядке. Если представить функции `scale2` и `translate1left` в виде машин, принимающих и возвращающих трехмерные модели (рис. 4.7), то их можно объединить, передавая выходные данные первой машины на вход второй.

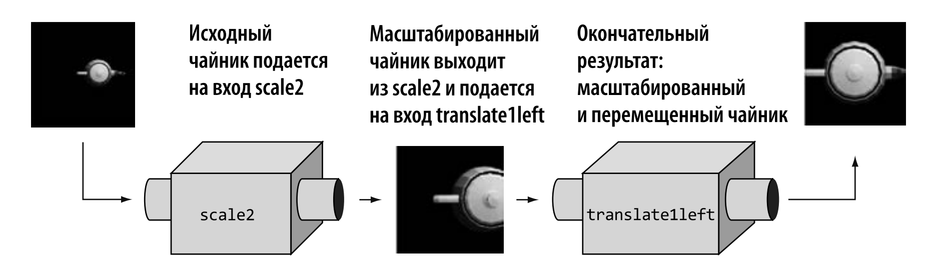


Рис. 4.7. Последовательное применение сначала scale2, а затем translate1left к модели чайника для получения преобразованной версии

Можно представить удаление промежуточного этапа путем стыковки выхода первой машины с входом второй (рис. 4.8).

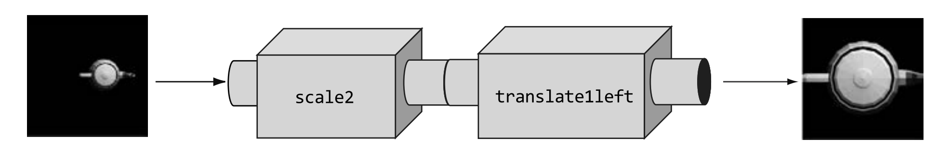


Рис. 4.8. Стыковка двух машин для получения новой машины, выполняющей два преобразования за один шаг

Результат такой стыковки можно рассматривать как создание новой машины, выполняющей всю работу за один шаг. Аналогичную стыковку можно выполнить и в программном коде. Мы можем написать универсальную функцию `compose`, принимающую две функции (например, функции векторных преобразований) и возвращающую новую функцию, которая является их композицией:

```
def compose(f1,f2):
    def new_function(input):
        return f1(f2(input))
    return new_function
```

Вместо определения самостоятельной функции `scale2_then_translate1left` можно записать:

```
scale2_then_translate1left = compose(translate1left, scale2)
```

Возможно, вы слышали, что функции в Python считаются первоклассными объектами. Под этими словами обычно подразумевается, что функции в языке Python могут присваиваться переменным, передаваться в параметрах другим функциям, создаваться на лету и возвращаться в виде возвращаемых значений. Эти приемы *функционального программирования* помогают создавать сложные программы, комбинируя новые функции из существующих.

В Интернете не первый год ведутся споры о том, насколько «кошерно» для Python функциональное программирование (или, как сказал бы фанат Python, является ли функциональное программирование питоническим). Я не буду рассуждать о стиле программирования, но использую приемы функционального программирования, потому что для нас центральными объектами изучения являются функции, в частности векторные преобразования. После знакомства с функцией `compose` я покажу вам еще несколько функциональных рецептов, оправдывающих это отступление. Все они реализованы в файле `transforms.py`.

Далее мы неоднократно будем применять векторное преобразование к каждой вершине в каждом треугольнике, определяющем трехмерную модель. Чтобы упростить эту задачу, можно написать функцию, а не всякий раз создавать заново генератор списка. Приведенная далее функция `polygon_map` принимает векторное преобразование и список многоугольников (обычно треугольников) и применяет преобразование к каждой вершине каждого многоугольника, получая новый список новых многоугольников:

```
def polygon_map(transformation, polygons):
    return [
        [transformation(vertex) for vertex in triangle]
        for triangle in polygons
    ]
```

С помощью этой вспомогательной функции можно применить `scale2` к исходному чайнику одной строкой:

```
draw_model(polygon_map(scale2, load_triangles()))
```

Обе функции, `compose` и `polygon_map`, принимают векторные преобразования в аргументах, но также было бы полезно иметь функции, возвращающие векторные преобразования. Например, кого-то может обеспокоить, что мы назвали функцию `scale2` и жестко задали коэффициент масштабирования 2 в ее определении.

Заменой ей могла бы быть функция `scale_by`, возвращающая преобразование масштабирования для заданного скаляра:

```
def scale_by(scalar):
    def new_function(v):
        return scale(scalar, v)
    return new_function
```

Имея такую функцию, можно сделать вызов `scale_by(2)` и получить новую функцию, которая ведет себя в точности как `scale2`. По аналогии с изображением функций в виде машин, получающих входные и выпускающих выходные данные, функцию `scale_by` тоже можно изобразить как машину, получающую числа и выдающую новые функциональные машины (рис. 4.9).

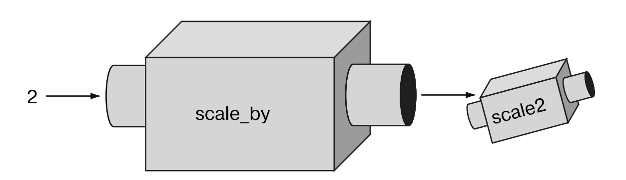


Рис. 4.9. Функциональная машина, получающая числа и выпускающая новые функциональные машины

В качестве упражнения напишите аналогичную функцию `translate_by`, которая принимает вектор параллельного переноса и возвращает функцию, выполняющую этот перенос. В функциональном программировании данный процесс называется *каррированием*. Суть каррирования заключается в использовании функции, принимающей несколько параметров, для создания функции, которая возвращает другие функции.

В результате получается программная машина, которая действует одинаково, но вызывается по-разному: например, `scale_by(s)(v)` дает тот же результат, что и `scale(s, v)`, для любых входных данных s и v. Преимущество в том, что `scale(...)` и `add(...)` принимают разные типы аргументов, поэтому результирующие функции, `scale_by(s)` и `translate_by(w)`, взаимозаменяемы. В дальнейшем мы применим этот же прием к поворотам: для любого заданного угла можно создать векторное преобразование, поворачивающее модель на этот угол.

4.1.3. Поворот объекта вокруг оси

Вы уже видели в главе 2, как повернуть двухмерную фигуру: нужно преобразовать декартовы координаты в полярные, изменить угол на величину поворота и выполнить обратное преобразование координат. Несмотря на то что этот прием применялся в двухмерном пространстве, он пригодится нам и в трехмерном, потому что все повороты трехмерных векторов в каком-то смысле изолированы

на плоскостях. Представьте, например, точку в трехмерном пространстве, поворачивающуюся вокруг оси z. Ее координаты x и y изменяются, но координата z остается прежней. Если точка вращается вокруг оси z, она остается на окружности с постоянной координатой z независимо от угла поворота (рис. 4.10).

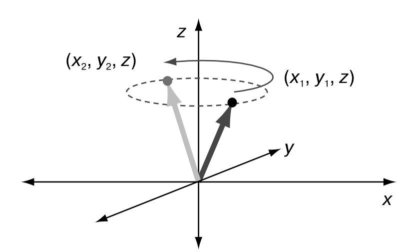


Рис. 4.10. Поворот точки вокруг оси *z*

Это означает, что трехмерную точку можно повернуть вокруг оси z, сохранив постоянной координату z и применив двухмерную функцию поворота только к координатам x и y. Код, который мы рассмотрим далее, можно найти также в файле `rotate_teapot.py`. Для начала напишем двухмерную функцию поворота, использующую стратегию, рассмотренную в главе 2:

```
def rotate2d(angle, vector):
    l,a = to_polar(vector)
    return to_cartesian((l, a+angle))
```

Эта функция принимает угол и двухмерный вектор и возвращает двухмерный вектор — результат поворота. Теперь напишем функцию `rotate_z`, которая применяет функцию `rotate2d` только к компонентам x и y трехмерного вектора:

```
def rotate_z(angle, vector):
    x,y,z = vector
    new_x, new_y = rotate2d(angle, (x,y))
    return new_x, new_y, z
```

Продолжая мыслить парадигмой функционального программирования, мы можем каррировать эту функцию. Независимо от угла поворота каррированная версия производит векторное преобразование, выполняющее соответствующий поворот:

```
def rotate_z_by(angle):
    def new_function(v):
        return rotate_z(angle,v)
    return new_function
```

Посмотрим, как она действует. Вызов

```
draw_model(polygon_map(rotate_z_by(pi/4.), load_triangles()))
```

рисует чайник, повернутый на угол $\pi/4$, или 45° (рис. 4.11).

Мы можем написать аналогичную функцию, поворачивающую чайник вокруг оси x, тогда поворот влияет только на координаты y и z вектора:

```
def rotate_x(angle, vector):
    x,y,z = vector
```

```
    new_y, new_z = rotate2d(angle, (y,z))
    return x, new_y, new_z
def rotate_x_by(angle):
    def new_function(v):
        return rotate_x(angle,v)
    return new_function
```

Функция `rotate_x_by` выполняет поворот вокруг оси x за счет фиксации координаты x и выполнения двухмерного поворота в плоскости yz. Код

```
draw_model(polygon_map(rotate_x_by(pi/2.), load_triangles()))
```

рисует чайник, повернутый на 90°, или $\pi/2$ рад (против часовой стрелки), вокруг оси x, в результате чего мы получаем вид чайника сверху (рис. 4.12).



Рис. 4.11. Чайник, повернутый на 45° против часовой стрелки вокруг оси z

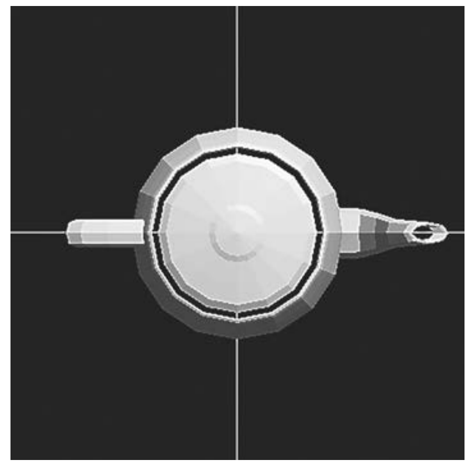


Рис. 4.12. Чайник, повернутый на угол $\pi/2$ вокруг оси x

Вы можете воспроизвести рис. 4.12, запустив сценарий `rotate_teapot_x.py`. Затенение повернутых чайников выполняется одинаково: их самые яркие полигоны находятся вверху и справа, что ожидаемо, так как источник света остается в точке (1, 2, 3). Это хороший признак того, что мы поворачиваем чайник, а не просто меняем точку местоположения наблюдателя, как раньше.

Как оказывается, можно выполнить *любой* поворот, какой пожелаем, объединив повороты вокруг осей x и z. В упражнениях в конце раздела вы сможете попробовать свои силы в выполнении других поворотов, а сейчас перейдем к другим видам векторных преобразований.

4.1.4. Изобретение своих геометрических преобразований

До сих пор я рассказывал о векторных преобразованиях, которые мы уже видели в предыдущих главах. Теперь предлагаю посмотреть, какие еще интересные преобразования можно придумать. Помните, единственное требование к преобразованиям трехмерных векторов таково: они должны принимать один трехмерный вектор и возвращать новый трехмерный вектор. Итак, рассмотрим несколько преобразований, не попадающих ни в одну из категорий, которые мы видели до сих пор.

Будем изменять координаты по одной. Следующая функция растягивает векторы в четыре раза, но только в направлении x:

```
def stretch_x(vector):
    x,y,z = vector
    return (4.*x, y, z)
```

В результате получается чайник, вытянутый вдоль оси x — от ручки к носику (рис. 4.13). Этот рисунок можно получить, запустив сценарий `stretch_teapot.py`.

Аналогичная функция `stretch_y` растягивает чайник по вертикали. У вас уже достаточно умений, чтобы реализовать и применить ее к чайнику самостоятельно. Должно получиться изображение, как на рис. 4.14. Если возникнут затруднения, то загляните в файл `stretch_teapot_y.py`.

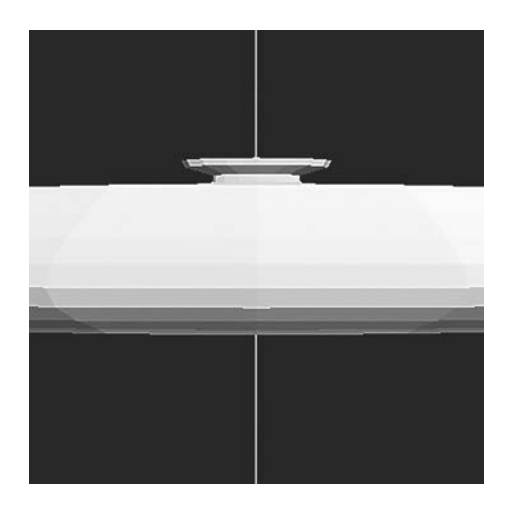


Рис. 4.13. Чайник, вытянутый вдоль оси *x*



Рис. 4.14. Чайник, вытянутый вдоль оси *y*

Мы можем проявить еще бóльшую изобретательность и растянуть чайник, увеличив координату y в кубе, а не просто умножив ее на некоторое число. Преобразование

```
def cube_stretch_z(vector):
    x,y,z = vector
    return (x, y*y*y, z)
```

приведет к тому, что у чайника будет непропорционально вытянутая крышка, как реализовано в `cube_teapot.py` и показано на рис. 4.15.

Если в формуле преобразования сложить две любые координаты из трех, например x и y, то можно получить наклоненный чайник. Это преобразование реализовано в `slant_teapot.py` и показано на рис. 4.16:

```
def slant_xy(vector):
    x,y,z = vector
    return (x+y, y, z)
```

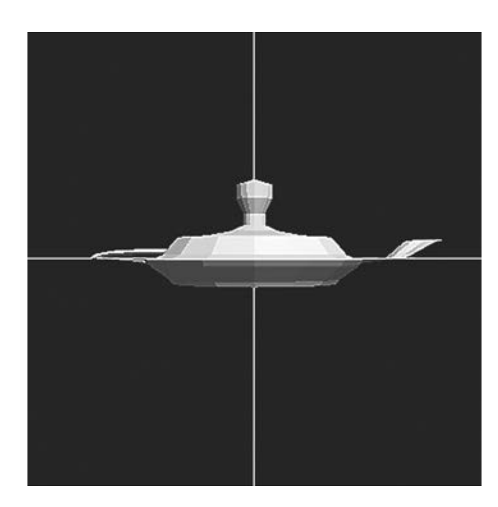


Рис. 4.15. Возведение вертикального измерения в куб

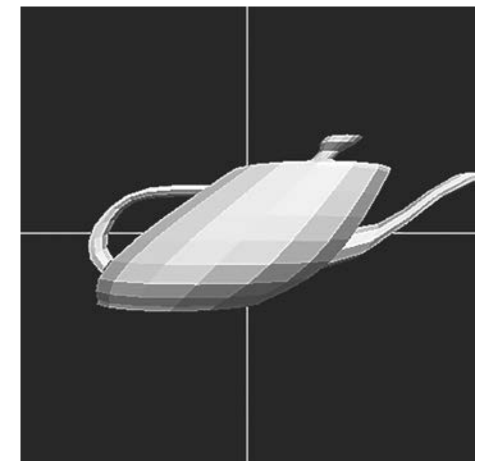


Рис. 4.16. Прибавление координаты *y* к координате *x* приводит к наклону чайника в направлении *x*

Дело не в важности или полезности любого из этих преобразований, а в том, что любое математическое преобразование векторов, образующих трехмерную модель, оказывает *некоторое* влияние на геометрическую форму модели. Преобразования можно довести до крайности и исказить модель до неузнаваемости. В действительности многие преобразования в целом не допускают такого, и мы классифицируем их в следующем разделе.

4.1.5. Упражнения

Упражнение 4.1. Реализуйте функцию `translate_by`, упомянутую в подразделе 4.1.2, которая принимает вектор переноса и возвращает функцию, применяющую его.

Решение.

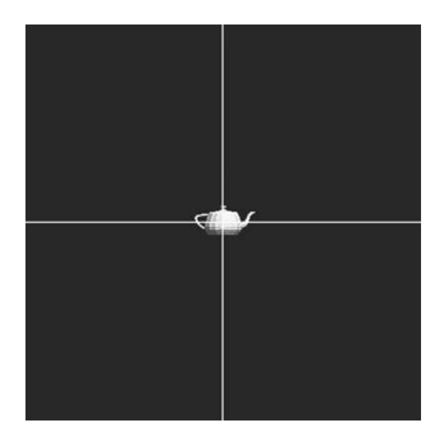
```
def translate_by(translation):
    def new_function(v):
        return add(translation,v)
    return new_function
```

Упражнение 4.2. Нарисуйте чайник, сдвинутый на 20 единиц в отрицательном направлении оси *z*. Как выглядит полученное изображение?

Решение. Желаемый результат можно получить, применив `translate_by((0, 0, -20))` к каждому вектору каждого многоугольника с помощью `polgyon_map`:

```
draw_model(polygon_map(translate_by((0,0,-20)), load_triangles()))
```

Напомню, что мы находимся на оси *z* в 5 единицах от чайника. Это преобразование отдалит чайник на 20 единиц от нас, поэтому он будет выглядеть меньше оригинала. Полную реализацию можно найти в файле `translate_teapot_down_z.py`.

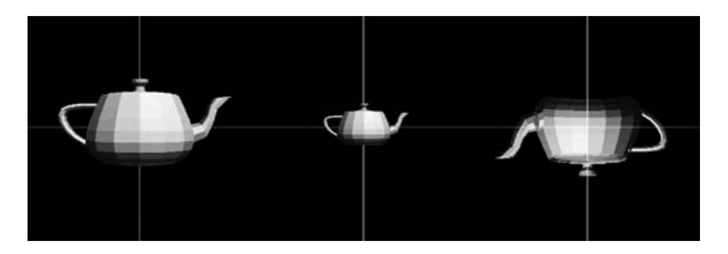


Чайник переместился на 20 единиц вдоль оси *z*. Он кажется меньше, потому что находится дальше от точки наблюдения

Упражнение 4.3. Мини-проект. Что произойдет с чайником, если каждый вектор масштабировать скаляром в диапазоне от 0 до 1? Что произойдет, если масштабировать его скаляром −1?

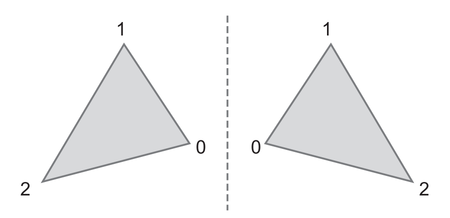
Решение. Чтобы узнать это, используйте `scale_by(0.5)` и `scale_by(-1)`:

```
draw_model(polygon_map(scale_by(0.5), load_triangles()))
draw_model(polygon_map(scale_by(-1), load_triangles()))
```



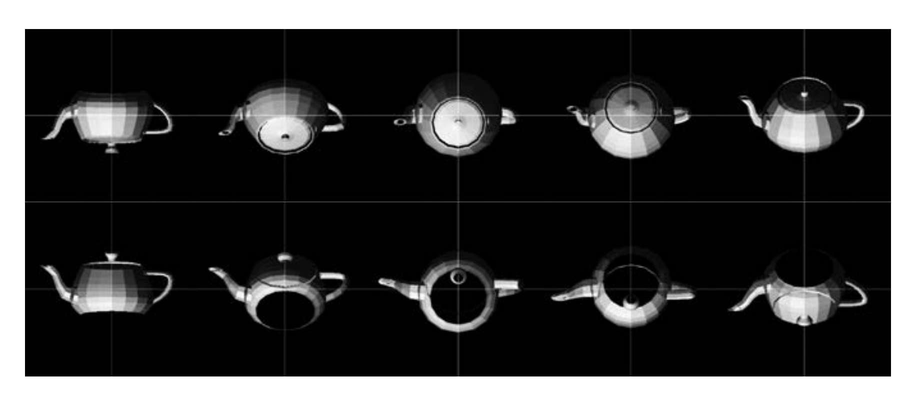
Слева направо: оригинальный чайник, результаты масштабирования на 0,5 и –1

Как видите, `scale_by(0.5)` уменьшает размеры чайника вдвое. Применение `scale_by(-1)`, похоже, поворачивает чайник на 180°, но на самом деле все немного сложнее. В действительности чайник выворачивается наизнанку! Каждый треугольник был отражен, поэтому каждый вектор нормали теперь указывает внутрь чайника, а не наружу от его поверхности.



Отражение меняет ориентацию треугольника. Индексированные вершины расположены в порядке против часовой стрелки слева и по часовой стрелке в отражении справа. Векторы нормали к этим треугольникам направлены в противоположные стороны

Поворачивая этот чайник, можно заметить, что он отрисовывается не совсем корректно. По этой причине необходимо проявлять осторожность, занимаясь отражением моделей!



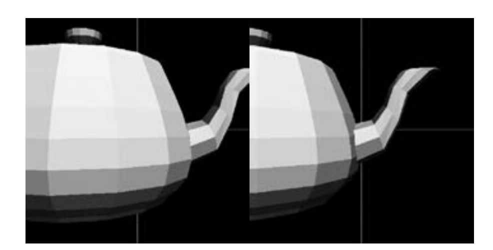
Повернутый чайник после его отражения выглядит несколько неправильно. Видны кое-какие элементы, которые должны быть скрыты. Например, внизу справа можно видеть и крышку, и прозрачное дно

Упражнение 4.4. Примените к чайнику сначала `translate1left`, а затем `scale2`. Изменится ли результат, если порядок функций будет обратным? Почему?

Решение. Мы можем объединить эти две функции в указанном порядке, а затем применить их с помощью `polygon_map`:

```
draw_model(polygon_map(compose(scale2, translate1left), load_triangles()))
```

В обоих случаях чайник увеличится в размерах в два раза, но в первом случае он сместится влево дальше. Это связано с тем, что при масштабировании после переноса на коэффициент 2 расстояние переноса также удваивается. Вы можете убедиться в этом, запустив сценарии `scale_translate_teapot.py` и `translate_scale_teapot.py` и сравнив результаты.



Масштабирование с последующим переносом *(слева)* и перенос с последующим масштабированием *(справа)*

Упражнение 4.5. Что получится в результате применения `compose(scale_by(0.4), scale_by(1.5))`?

Решение. В результате применения этой комбинации преобразований вектор будет масштабирован сначала с коэффициентом 1,5, а затем 0,4, что равносильно масштабированию с коэффициентом 0,6. Полученная фигура уменьшится в размерах до 60 % от размеров оригинала.

Упражнение 4.6. Измените функцию `compose(f, g)` так, чтобы она имела сигнатуру `compose(*args)`. То есть новая функция должна принимать неограниченное количество функций преобразования и возвращать новую функцию, являющуюся их композицией.

Решение

```
def compose(*args):
    def new_function(input):
        state = input
        for f in reversed(args):
            state = f(state)
        return state
    return new_function
```

Начало определения новой функции, которую должна вернуть compose

Установить текущее состояние равным входному аргументу input

На каждом шаге обновлять состояние применением следующей функции. Конечное состояние формируется применением всех функций в правильном порядке

Выполнить обход входных функций в args в обратном порядке, потому что самая последняя функция в композиции применяется первой. Например, вызов compose(f,g,h)(x) должен действовать так же, как f(g(h(x))), то есть первой должна применяться функция h

Для проверки можно создать несколько функций и применить их с помощью новой функции `compose`:

```
def prepend(string):
    def new_function(input):
        return string + input
    return new_function

f = compose(prepend("P"), prepend("y"), prepend("t"))
```

Вызов `f("hon")` должен вернуть строку `"Python"`. То есть новая сконструированная функция `f` должна добавлять строку `"Pyt"` к любой заданной строке.

Упражнение 4.7. Напишите функцию `curry2(f)`, которая принимает функцию `f(x,y)` двух аргументов и возвращает каррированную версию. Например, для случая `g = curry2(f)` вызовы `f(x,y)` и `g(x)(y)` должны вернуть один и тот же результат.

Решение. Функция `curry2(f)` должна вернуть новую функцию, которая, в свою очередь, создает еще одну новую функцию:

```
def curry2(f):
    def g(x):
        def new_function(y):
            return f(x,y)
        return new_function
    return g
```

С помощью этой функции мы могли бы сконструировать функцию `scale_by` так:

```
>>> scale_by = curry2(scale)
>>> scale_by(2)((1,2,3))

(2, 4, 6)
```

Упражнение 4.8. Определите результат преобразования `compose(rotate_z_by(pi/2), rotate_x_by(pi/2))` без запуска. Что получится, если поменять порядок функций в композиции?

Решение. Эта композиция выполнит поворот по часовой стрелке на $\pi/2$ вокруг оси y. Если те же функции применить в обратном порядке, будет выполнен поворот против часовой стрелки на $\pi/2$ вокруг оси y.

Упражнение 4.9. Напишите функцию `stretch_x(scalar, vector)`, которая масштабирует целевой вектор с заданным коэффициентом, но только вдоль оси x. Напишите также каррированную версию `stretch_x_by`, чтобы вызов `stretch_x_by(scalar)(vector)` возвращал тот же результат.

Решение

```
def stretch_x(scalar,vector):
    x,y,z = vector
    return (scalar*x, y, z)

def stretch_x_by(scalar):
    def new_function(vector):
        return stretch_x(scalar,vector)
    return new_function
```

4.2. ЛИНЕЙНЫЕ ПРЕОБРАЗОВАНИЯ

Типичные векторные преобразования, на которых мы сосредоточимся, называются *линейными преобразованиями*. Это еще один объект изучения линейной алгебры наряду с векторами. Линейными называют такие преобразования, до и после которых векторная арифметика выглядит одинаково. Рассмотрим несколько графиков, чтобы понять, что это значит.

4.2.1. Сохраняющая векторная арифметика

Двумя наиболее важными арифметическими операциями над векторами являются сложение и умножение на скаляр. Вернемся к двухмерным представлениям этих операций и посмотрим, как они выглядят до и после преобразования.

Сумму двух векторов можно изобразить как новый вектор, который получается, когда один вектор начинается там, где заканчивается другой по правилу параллелограмма. Например, на рис. 4.17 представлена векторная сумма **u** + **v** = **w**.

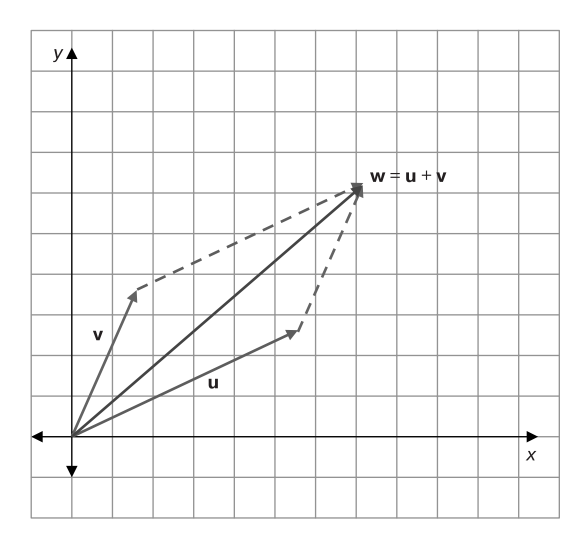


Рис. 4.17. Геометрическая демонстрация векторной суммы **u** + **v** = **w**

Вопрос заключается в следующем: если применить одно и то же векторное преобразование ко всем трем векторам на этом графике, будет ли результат выглядеть похожим на сумму векторов? Попробуем выполнить поворот вокруг начала координат против часовой стрелки и назовем это преобразование R. На рис. 4.18 показан результат поворота векторов **u**, **v** и **w** на один и тот же угол.

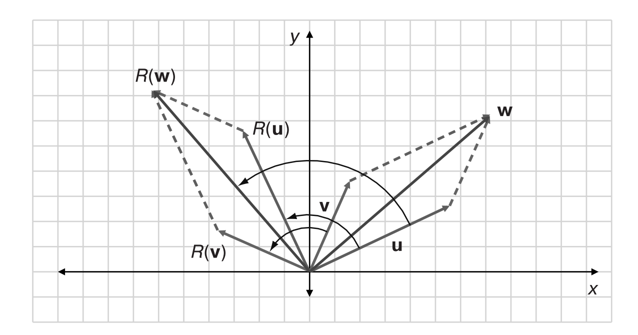


Рис. 4.18. При повороте векторов **u**, **v** и **w** на один и тот же угол сумма сохраняет свои свойства

Результат поворота представляет векторную сумму $R(\mathbf{u}) + R(\mathbf{v}) = R(\mathbf{w})$. Вы можете выбрать любые три вектора **u**, **v** и **w**, и если $\mathbf{u} + \mathbf{v} = \mathbf{w}$, то при применении одного и того же преобразования поворота R к каждому из векторов обнаружите, что $R(\mathbf{u}) + R(\mathbf{v}) = R(\mathbf{w})$. Описывая это свойство, мы говорим, что поворот *сохраняет* векторные суммы.

Точно так же поворот сохраняет результат умножения вектора на скаляр. Если **v** — вектор, а $s\mathbf{v}$ — произведение **v** на скаляр s, то $s\mathbf{v}$ будет указывать в том же направлении, что и **v**, но длина нового вектора будет кратна длине вектора **v** с коэффициентом s. Если выполнить поворот R векторов **v** и $s\mathbf{v}$ на один и тот же угол, то мы увидим, что результат $R(s\mathbf{v})$ — это произведение $R(\mathbf{v})$ на скаляр s (рис. 4.19).

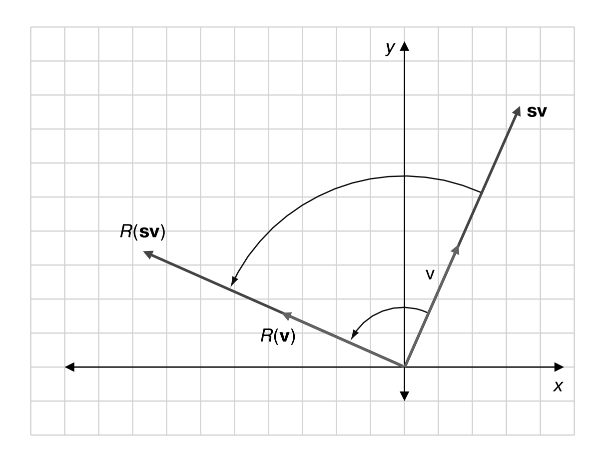


Рис. 4.19. Результат умножения на скаляр сохраняется после поворота

Повторю еще раз, что это только наглядный пример, а не доказательство, но какой бы вектор **v**, скаляр s и угол поворота R вы ни взяли, общая картина

сохранится. Повороты и любые другие векторные преобразования, сохраняющие векторную сумму и результат умножения на скаляр, называются *линейными преобразованиями*.

ЛИНЕЙНОЕ ПРЕОБРАЗОВАНИЕ

Линейное преобразование — это векторное преобразование T, сохраняющее векторную сумму и результат умножения на скаляр. То есть для любой пары векторов **u** и **v**, а также для любого вектора **v** и скаляра s выполняются условия

$$T(\mathbf{u}) + T(\mathbf{v}) = T(\mathbf{u} + \mathbf{v})$$

и

$$T(s\mathbf{v}) = sT(\mathbf{v}).$$

Теперь сделайте паузу, чтобы переварить это определение, — линейные преобразования настолько важны, что в их честь назван весь предмет линейная алгебра. Чтобы вы могли отличать линейные преобразования от других при встрече с ними, рассмотрим еще несколько примеров.

4.2.2. Изображение линейных преобразований

Рассмотрим контрпример — векторное преобразование, *не* являющееся линейным. Таковым может служить преобразование $S(\mathbf{v})$, которое принимает вектор $\mathbf{v} = (x, y)$ и дает в результате вектор с координатами, являющимися квадратами исходных координат: $S(\mathbf{v}) = (x^2, y^2)$. Возьмем, например, сумму $\mathbf{u} = (2, 3)$ и $\mathbf{v} = (1, -1)$. Сумма $(2, 3) + (1, -1) = (3, 2)$. Это демонстрирует схема сложения векторов на рис. 4.20.

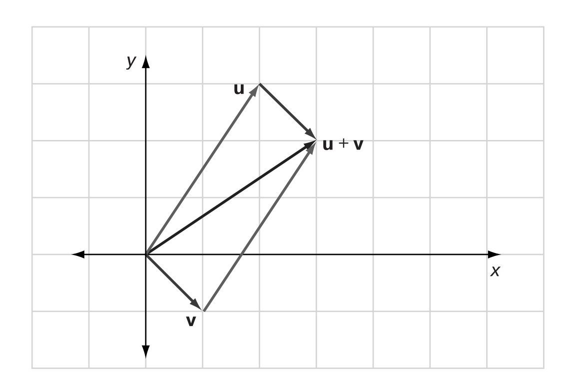


Рис. 4.20. Сумма векторов **u** = (2, 3) и **v** = (1, –1), **u** + **v** = (3, 2)

Теперь применим преобразование S к каждому из них: $S(\mathbf{u}) = (4, 9)$, $S(\mathbf{v}) = (1, 1)$ и $S(\mathbf{u} + \mathbf{v}) = (9, 4)$. На рис. 4.21 ясно видно, что $S(\mathbf{u}) + S(\mathbf{v})$ не равно $S(\mathbf{u} + \mathbf{v})$.

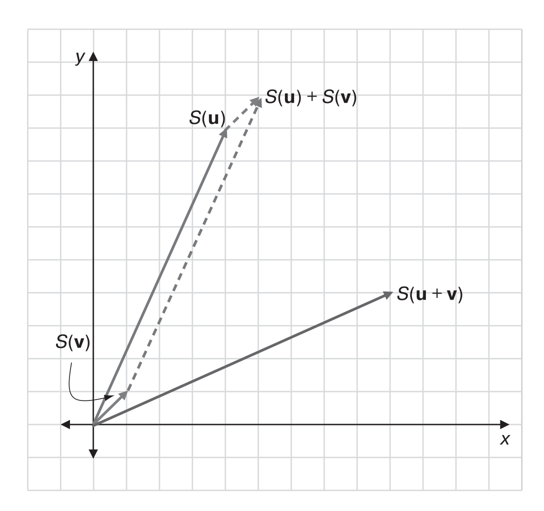


Рис. 4.21. Преобразование S не сохраняет сумму! $S(\mathbf{u}) + S(\mathbf{v})$ намного больше, чем $S(\mathbf{u} + \mathbf{v})$

В качестве упражнения можете попробовать найти собственный контрпример, демонстрирующий преобразование S, которое тоже не сохраняет кратность скаляру. А пока рассмотрим еще одно преобразование. Пусть $D(\mathbf{v})$ — векторное преобразование, масштабирующее исходный вектор с коэффициентом 2. То есть $D(\mathbf{v}) = 2\mathbf{v}$. Это преобразование *сохраняет* векторные суммы: если $\mathbf{u} + \mathbf{v} = \mathbf{w}$, то $2\mathbf{u} + 2\mathbf{v} = 2\mathbf{w}$. Наглядный пример показан на рис. 4.22.

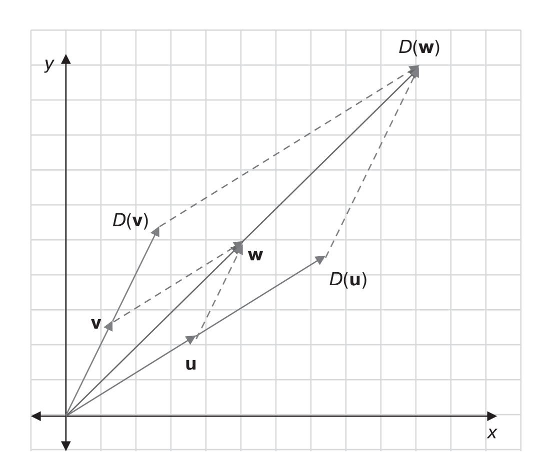


Рис. 4.22. Удвоение длин исходных векторов сохраняет их сумму: если $\mathbf{u} + \mathbf{v} = \mathbf{w}$, то $D(\mathbf{u}) + D(\mathbf{v}) = D(\mathbf{w})$

Точно так же преобразование $D(\mathbf{v})$ сохраняет результат умножения на скаляр. Показать визуально это немного сложнее, но проверку можно выполнить и алгебраически. Для любого скаляра s выполняется условие $D(s\mathbf{v}) = 2(s\mathbf{v}) = s(2\mathbf{v}) = sD(\mathbf{v})$.

А можно ли назвать линейным преобразованием параллельный перенос? Пусть $B(\mathbf{v})$ переносит любой данный вектор $\mathbf{v}$ на $(7, 0)$. Может показаться странным, но это *не* линейное преобразование. На рис. 4.23 показан наглядный контрпример, где $\mathbf{u} + \mathbf{v} = \mathbf{w}$, но $B(\mathbf{v}) + B(\mathbf{w}) \neq B(\mathbf{v} + \mathbf{w})$.

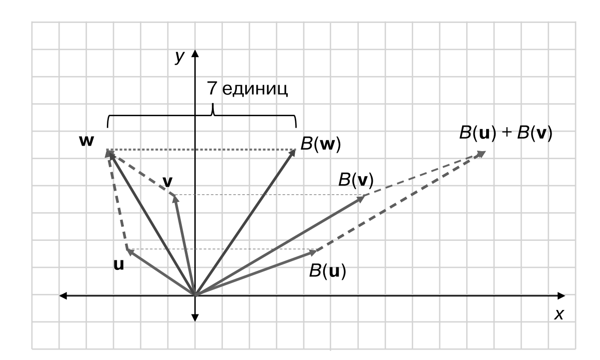


Рис. 4.23. Параллельный перенос B не сохраняет векторную сумму, потому что $B(\mathbf{v}) + B(\mathbf{w}) \neq B(\mathbf{v} + \mathbf{w})$

Оказывается, чтобы преобразование было линейным, оно не должно перемещать начало координат (причина описывается далее в упражнениях). Параллельный перенос с любым вектором ненулевой длины преобразует начало координат, которое оказывается в другой точке, поэтому он не может быть линейным преобразованием.

В числе других примеров линейных преобразований можно назвать отражение, проекцию, сдвиг и любые трехмерные аналоги предыдущих линейных преобразований. Они определены в разделе с упражнениями, где вам будет представлено несколько примеров и предложено проверить линейность каждого из преобразований, проверив, сохраняют ли они векторную сумму и результат умножения на скаляр. Попрактиковавшись, вы сможете отличать линейные преобразования от нелинейных. Далее мы рассмотрим полезные свойства линейных преобразований.

4.2.3. Полезные свойства линейных преобразований

Линейные преобразования сохраняют векторные суммы и результат умножения на скаляр, а также более широкий класс арифметических векторных операций. Наиболее общая операция называется *линейной комбинацией*. Линейная комбинация набора векторов — это сумма произведений векторов на скаляры. Например, $3\mathbf{u} - 2\mathbf{v}$ — одна из линейных комбинаций двух векторов, $\mathbf{u}$ и $\mathbf{v}$. Выражение

$0{,}5\mathbf{u} - \mathbf{v} + 6\mathbf{w}$ — это линейная комбинация трех векторов, **u**, **v** и **w**. Поскольку линейные преобразования сохраняют векторные суммы и результат умножения на скаляр, они сохраняют и результат линейных комбинаций.

Этот факт можно сформулировать алгебраически. Если есть набор из n векторов, $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n$, а также набор из n скаляров, $s_1, s_2, s_3, ..., s_n$, то линейное преобразование T сохраняет линейную комбинацию:

$$T(s_1\mathbf{v}_1 + s_2\mathbf{v}_2 + s_3\mathbf{v}_3 + ... + s_n\mathbf{v}_n) = s_1T(\mathbf{v}_1) + s_2T(\mathbf{v}_2) + s_3T(\mathbf{v}_3) + ... + s_nT(\mathbf{v}_n).$$

Одна из простых для отображения линейных комбинаций, которую мы уже видели, — это $(1/2)\mathbf{u} + (1/2)\mathbf{v}$, что эквивалентно $(1/2)(\mathbf{u} + \mathbf{v})$. На рис. 4.24 показано, что эта линейная комбинация двух векторов дает нам середину соединяющего их отрезка.

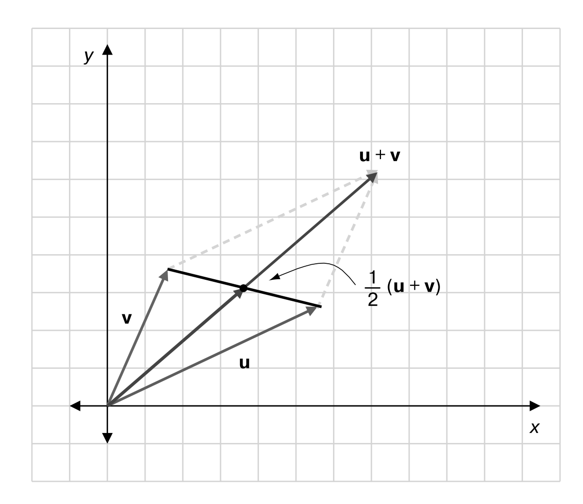


Рис. 4.24. Середину отрезка, соединяющего два вектора, **u** и **v**, можно найти как линейную комбинацию $(1/2)\mathbf{u} + (1/2)\mathbf{v} = (1/2)(\mathbf{u} + \mathbf{v})$

Это означает, что линейные преобразования сохраняют также средние точки: например, $T((1/2)\mathbf{u} + (1/2)\mathbf{v}) = (1/2)T(\mathbf{u}) + (1/2)T(\mathbf{v})$, что является серединой отрезка, соединяющего $T(\mathbf{u})$ и $T(\mathbf{v})$, как показано на рис. 4.25.

Линейная комбинация $0{,}25\mathbf{u} + 0{,}75\mathbf{v}$ тоже лежит на отрезке, соединяющем **u** и **v** (рис. 4.26). В частности, эта точка находится на 75 % пути от **u** до **v**. Аналогично, $0{,}6\mathbf{u} + 0{,}4\mathbf{v}$ — это 40 % пути от **u** до **v** и т. д.

На самом деле каждая точка на отрезке между двумя векторами представляет собой взвешенное значение в форме $s\mathbf{u} + (1 - s)\mathbf{v}$ для некоторого числа s между 0 и 1. В качестве доказательства на рис. 4.27 показаны векторы $s\mathbf{u} + (1 - s)\mathbf{v}$ для $\mathbf{u} = (-1, 1)$ и $\mathbf{v} = (3, 4)$, соответствующие 10 значениям s от 0 до 1, и затем соответствующие 100 значениям s от 0 до 1.

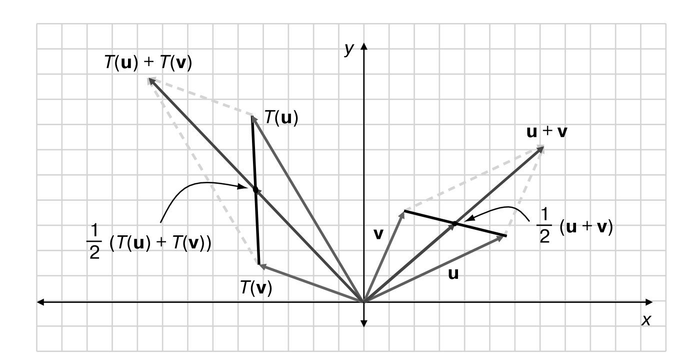


Рис. 4.25. Точка на середине отрезка, соединяющего два вектора, — это линейная комбинация векторов, поэтому линейное преобразование *T* сохраняет точку на середине отрезка между *T*(**u**) и *T*(**v**)

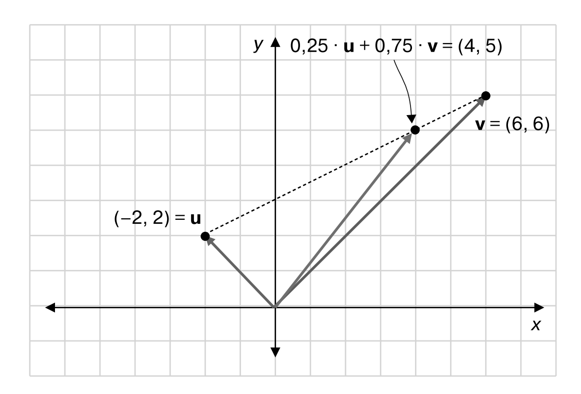


Рис. 4.26. Точка 0,25**u** + 0,75**v** лежит на отрезке, соединяющем u и v, на 75 % пути от **u** до **v**. В этом можно убедиться на примере векторов **u** = (–2, 2) и **v** = (6, 6)

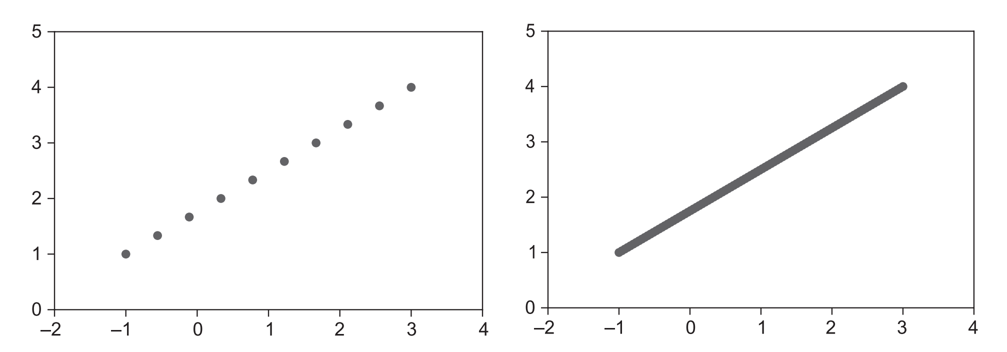


Рис. 4.27. Различные взвешенные значения для векторов (–1, 1) и (3, 4): *слева* — 10 значений *s* от 0 до 1; *справа* — 100 значений *s* от 0 до 1

Суть здесь в том, что каждая точка на отрезке, соединяющем векторы **u** и **v**, является взвешенным значением и, следовательно, линейной комбинацией точек **u** и **v**. Учитывая это, можно предположить, что отрезок целиком — это линейное преобразование.

Любая точка на отрезке, соединяющем **u** и **v**, является взвешенным значением **u** и **v** и выражается как $s \cdot \mathbf{u} + (1 - s) \cdot \mathbf{v}$ для некоторого значения s. Линейное преобразование T преобразует **u** и **v** в некоторые новые векторы $T(\mathbf{u})$ и $T(\mathbf{v})$. Точка на отрезке преобразуется в некоторую новую точку $T(s \cdot \mathbf{u} + (1 - s) \cdot \mathbf{v})$ или $s \cdot T(\mathbf{u}) + (1 - s) \cdot T(\mathbf{v})$, которая, в свою очередь, является взвешенным значением $T(\mathbf{u})$ и $T(\mathbf{v})$ и, соответственно, лежит на отрезке, соединяющем $T(\mathbf{u})$ и $T(\mathbf{v})$, как показано на рис. 4.28.

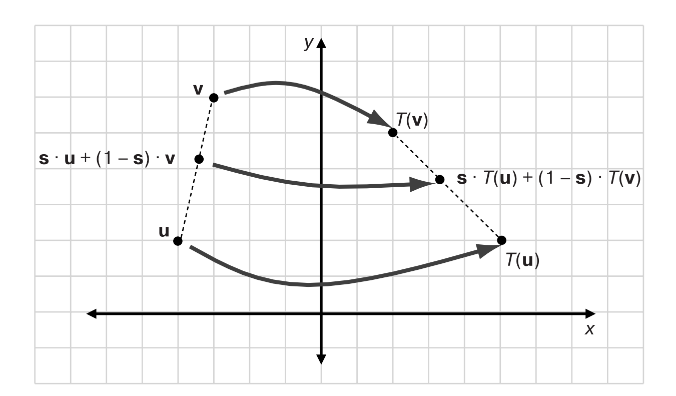


Рис. 4.28. Линейное преобразование *T* преобразует взвешенные значения **u** и **v** во взвешенные значения *T*(**u**) и *T*(**v**). Исходное взвешенное значение лежит на отрезке, соединяющем **u** и **v**, а преобразованное — на отрезке, соединяющем *T*(**u**) и *T*(**v**)

Как следствие, линейное преобразование T отображает каждую точку на отрезке, соединяющем **u** и **v**, в точку на отрезке, соединяющем $T(\mathbf{u})$ и $T(\mathbf{v})$. Это ключевое свойство линейных преобразований: они отображают каждый существующий отрезок в новый отрезок. Наши трехмерные модели состоят из многоугольников, которые ограничены отрезками, поэтому можно ожидать, что линейные преобразования в некоторой степени будут сохранять структуру моделей (рис. 4.29).

Нелинейное преобразование $S(\mathbf{v})$, преобразующее $\mathbf{v} = (x, y)$ в (x^2, y^2), напротив, искажает отрезки. Это означает, что треугольник, определяемый векторами **u**, **v** и **w**, на самом деле не переносится в другой треугольник, определяемый векторами $S(\mathbf{u})$, $S(\mathbf{v})$ и $S(\mathbf{w})$, как показано на рис. 4.30.

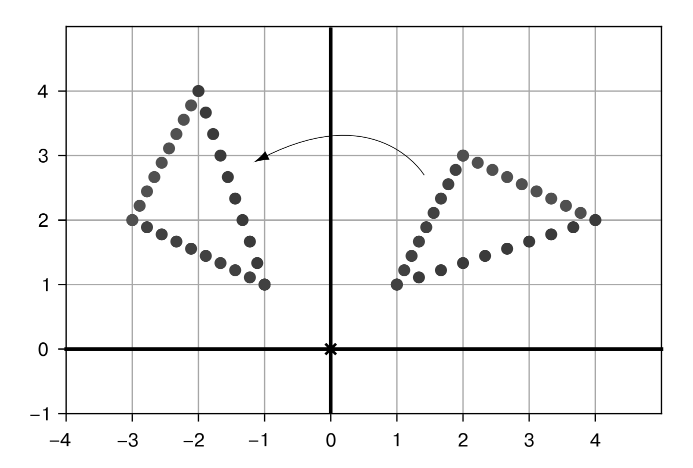


Рис. 4.29. Применение линейного преобразования (поворот на 60°) к точкам, образующим треугольник. В результате получается повернутый треугольник (*слева*)

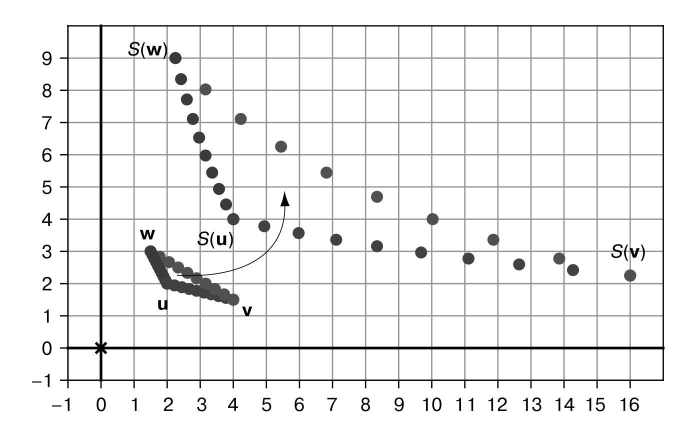


Рис. 4.30. Нелинейное преобразование *S* не сохраняет прямолинейность сторон треугольника

Таким образом, линейные преобразования учитывают алгебраические свойства векторов, сохраняя суммы, произведение на скаляр и линейные комбинации. Они также учитывают геометрические свойства наборов векторов, отображая отрезки и многоугольники, определяемые векторами, в новые отрезки и многоугольники, определяемые преобразованными векторами. Далее мы увидим, что линейные преобразования характеризуются не только геометрическими свойствами — они также легко вычисляются.

4.2.4. Вычисление результатов линейных преобразований

В главах 2 и 3 вы видели, как разложить двух- и трехмерные векторы на компоненты. Например, вектор (4, 3, 5) можно представить как сумму (4, 0, 0) + + (0, 3, 0) + (0, 0, 5). Это позволяет увидеть, как далеко протянется вектор в каждом из трех измерений в пространстве. Однако мы можем пойти еще дальше и разложить вектор на линейную комбинацию (рис. 4.31):

$$(4, 3, 5) = 4 \cdot (1, 0, 0) + 3 \cdot (0, 1, 0) + 5 \cdot (0, 0, 1).$$

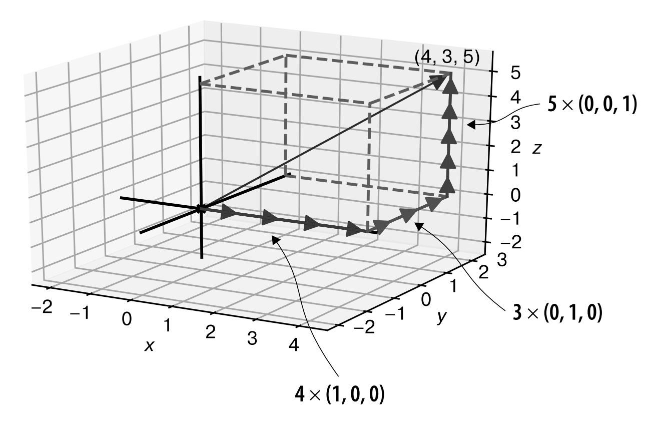


Рис. 4.31. Трехмерный вектор (4, 3, 5) как линейная комбинация векторов (1, 0, 0), (0, 1, 0) и (0, 0, 1)

Данный факт может показаться ничем не примечательным, и все же это одно из важнейших открытий линейной алгебры: любой трехмерный вектор можно разложить на линейную комбинацию трех векторов (1, 0, 0), (0, 1, 0) и (0, 0, 1). Скаляры, присутствующие в этом разложении, в точности совпадают с координатами вектора.

Три вектора, (1, 0, 0), (0, 1, 0) и (0, 0, 1), называются *стандартным базисом* трехмерного пространства. Они обозначаются $\mathbf{e}_1$, $\mathbf{e}_2$ и $\mathbf{e}_3$, поэтому предыдущую линейную комбинацию можно записать как (3, 4, 5) = $3\mathbf{e}_1 + 4\mathbf{e}_2 + 5\mathbf{e}_3$. В двухмерном пространстве используется базис $\mathbf{e}_1 = (1, 0)$ и $\mathbf{e}_2 = (0, 1)$, например, $(7, -4) = 7\mathbf{e}_1 - 4\mathbf{e}_2$ (рис. 4.32). (При упоминании $\mathbf{e}_1$ может подразумеваться (1, 0) или (1, 0, 0), но обычно это ясно из контекста, когда заранее известно, с каким пространством мы работаем — двух- или трехмерным.)

Мы просто записали одни и те же векторы немного по-другому, но оказалось, что это изменение упрощает вычисление линейных преобразований. Линейные преобразования учитывают линейные комбинации, поэтому для вычисления линейного преобразования достаточно лишь знать, как оно влияет на векторы стандартного базиса.

Рассмотрим наглядный пример (рис. 4.33). Пусть есть двухмерное векторное преобразование T, о котором мы ничего не знаем, кроме того, что оно линейное, и есть результат этого преобразования $T(\mathbf{e}_1)$ и $T(\mathbf{e}_2)$.

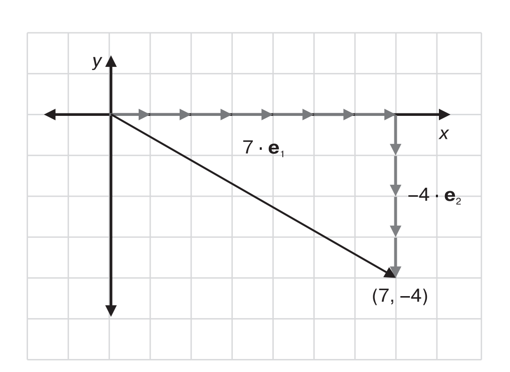


Рис. 4.32. Двухмерный вектор (7, –4) как линейная комбинация векторов стандартного базиса $\mathbf{e}_1$ и $\mathbf{e}_2$

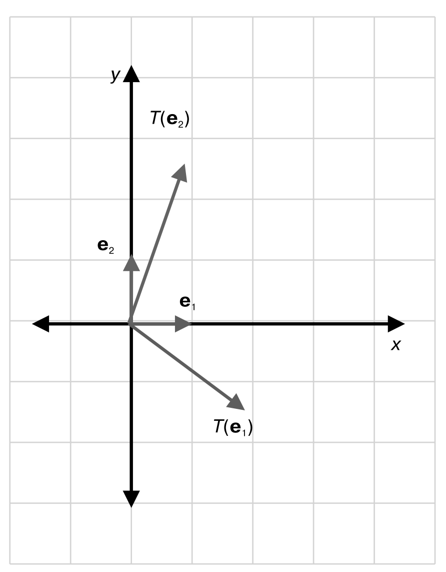


Рис. 4.33. Когда линейное преобразование применяется к векторам стандартного базиса в двухмерном пространстве, получаются два новых вектора

Для любого другого вектора $\mathbf{v}$ мы легко можем определить, где заканчивается $T(\mathbf{v})$. Например, пусть $\mathbf{v} = (3, 2)$, тогда можно утверждать:

$$T(\mathbf{v}) = T(3\mathbf{e}_1 + 2\mathbf{e}_2) = 3T(\mathbf{e}_1) + 2T(\mathbf{e}_2).$$

Зная, где находятся $T(\mathbf{e}_1)$ и $T(\mathbf{e}_2)$, можно найти $T(\mathbf{v})$, как показано на рис. 4.34.

Для большей конкретики рассмотрим пример в трехмерном пространстве. Пусть есть A — линейное преобразование, о котором известно только, что $A(\mathbf{e}_1) = (1, 1, 1)$, $A(\mathbf{e}_2) = (1, 0, -1)$ и $A(\mathbf{e}_3) = (0, 1, 1)$, и есть $\mathbf{v} = (-1, 2, 2)$. Нужно определить

результат $A(\mathbf{v})$. Сначала разложим $\mathbf{v}$ на линейную комбинацию трех векторов стандартного базиса. Поскольку $\mathbf{v} = (-1, 2, 2) = -\mathbf{e}_1 + 2\mathbf{e}_2 + 2\mathbf{e}_3$, можем сделать подстановку:

$$A(\mathbf{v}) = A(-\mathbf{e}_1 + 2\mathbf{e}_2 + 2\mathbf{e}_3).$$

Затем используем тот факт, что A — линейное преобразование и сохраняет линейные комбинации:

$$= -A(\mathbf{e}_1) + 2A(\mathbf{e}_2) + 2A(\mathbf{e}_3).$$

Наконец, выполним подстановку известных значений $A(\mathbf{e}_1)$, $A(\mathbf{e}_2)$ и $A(\mathbf{e}_3)$ и упростим:

$$= -(1, 1, 1) + 2 \cdot (1, 0, -1) + 2 \cdot (0, 1, 1) = (1, 1, -1).$$

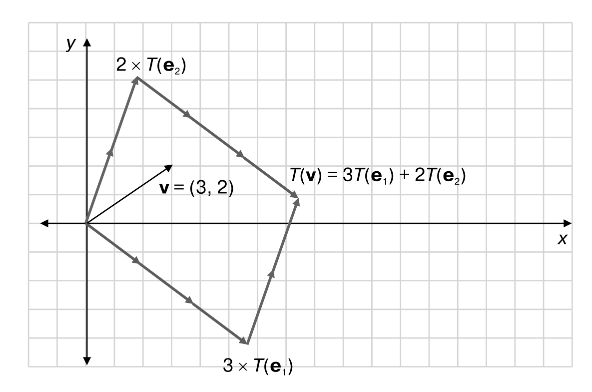


Рис. 4.34. Результат преобразования $T(\mathbf{v})$ можно вычислить для любого вектора $\mathbf{v}$ как линейную комбинацию $T(\mathbf{e}_1)$ и $T(\mathbf{e}_2)$

В качестве доказательства того, что мы действительно понимаем, как работает преобразование A, применим его к чайнику:

```
Ae1 = (1,1,1)
Ae2 = (1,0,-1)
Ae3 = (0,1,1)

def apply_A(v):
    return add(
        scale(v[0], Ae1),
        scale(v[1], Ae2),
        scale(v[2], Ae3)
    )

draw_model(polygon_map(apply_A, load_triangles()))
```

Известный результат применения A к векторам стандартного базиса

Функция apply_A(v) возвращает результат применения A к входному вектору v

Результат должен быть линейной комбинацией этих векторов, в которой скаляры являются координатами целевого вектора

Вызывает polygon_map, чтобы применить A к каждому вектору каждого треугольника в чайнике

Результат применения этого преобразования показан на рис. 4.35.



Рис. 4.35. Как видите, в этом повернутом и перекошенном результате преобразования отсутствует дно!

Итак, мы выяснили, что двухмерное линейное преобразование T полностью определяется значениями $T(\mathbf{e}_1)$ и $T(\mathbf{e}_2)$ — двумя векторами или четырьмя числами. Точно так же трехмерное линейное преобразование T полностью определяется значениями $T(\mathbf{e}_1)$, $T(\mathbf{e}_2)$ и $T(\mathbf{e}_3)$ — тремя векторами или девятью числами. В пространстве любой мерности поведение линейного преобразования определяется списком векторов или массивом массивов чисел. Такой массив массивов называется *матрицей*, и в следующей главе вы узнаете, как их использовать.

4.2.5. Упражнения

Упражнение 4.10. Вернемся к векторному преобразованию S, которое возводит в квадрат все координаты. Покажите алгебраически, что $S(s\mathbf{v}) = sS(\mathbf{v})$ выполняется не для всех скаляров s и двухмерных векторов $\mathbf{v}$.

Решение. Пусть $\mathbf{v} = (x, y)$. Тогда $s\mathbf{v} = (sx, sy)$ и $S(s\mathbf{v}) = (s^2x^2, s^2y^2) = s^2 \cdot (x^2, y^2) = s^2 \cdot S(\mathbf{v})$. Для большинства значений s и векторов $\mathbf{v}$ результат $S(s\mathbf{v}) = s^2 \cdot S(\mathbf{v})$ не равен $s \cdot S(\mathbf{v})$. Конкретным контрпримером может служить $s = 2$ и $\mathbf{v} = (1, 1, 1)$, где $S(s\mathbf{v}) = (4, 4, 4)$, а $s \cdot S(\mathbf{v}) = (2, 2, 2)$. Этот контрпример показывает, что S — не линейное преобразование.

Упражнение 4.11. Пусть T — векторное преобразование и $T(\mathbf{0}) \neq 0$, где $\mathbf{0}$ — это вектор, все координаты которого равны нулю. Почему T не является линейным по определению?

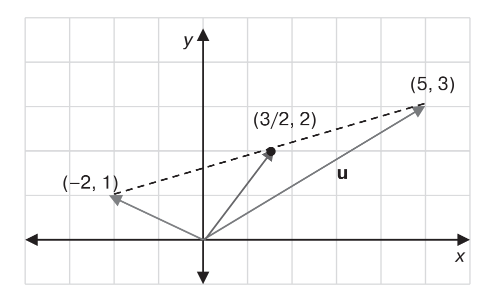
Решение. Для любого вектора $\mathbf{v}$ сумма $\mathbf{v} + \mathbf{0} = \mathbf{v}$. Чтобы преобразование T сохранило сумму векторов, должно выполняться условие $T(\mathbf{v} + \mathbf{0}) = T(\mathbf{v}) + T(\mathbf{0})$. Поскольку $T(\mathbf{v} + \mathbf{0}) = T(\mathbf{v})$, требуется, чтобы $T(\mathbf{v}) = T(\mathbf{v}) + T(\mathbf{0})$ или $\mathbf{0} = T(\mathbf{0})$. Учитывая, что это не так, T не может быть линейным преобразованием.

Упражнение 4.12. *Тождественное преобразование* — это векторное преобразование, возвращающее тот же вектор, который был передан на вход. Такое преобразование обозначается прописной буквой I, поэтому его определение можно записать как $I(\mathbf{v}) = \mathbf{v}$ для всех векторов $\mathbf{v}$. Объясните, почему I — это линейное преобразование.

Решение. Для любых векторов $\mathbf{v}$ и $\mathbf{w}$ выполняется равенство $I(\mathbf{v} + \mathbf{w}) = \mathbf{v} + \mathbf{w} = I(\mathbf{v}) + I(\mathbf{w})$. То же верно для любого скаляра s: $I(s\mathbf{v}) = s\mathbf{v} = s \cdot I(\mathbf{v})$. Эти равенства показывают, что тождественное преобразование сохраняет векторные суммы и произведения на скаляр.

Упражнение 4.13. Найдите координаты точки на середине отрезка, соединяющего (5, 3) и (−2, 1). Нарисуйте график со всеми тремя точками, чтобы проверить свою правоту.

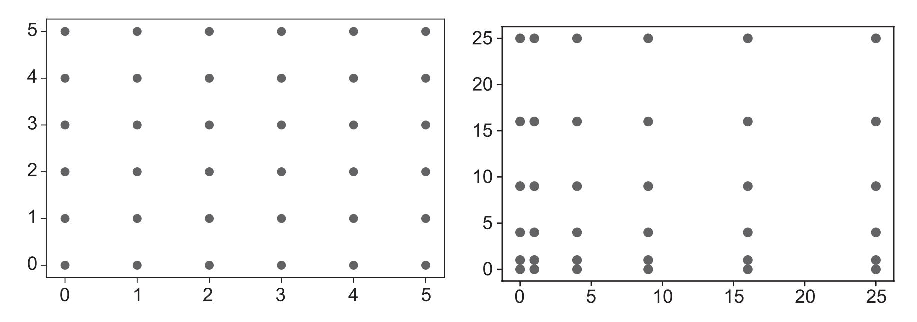
Решение. Искомая точка имеет координаты $(1/2)(5, 3) + (1/2)(-2, 1) = (5/2, 3/2) + (-1, 1/2) = (3/2, 2)$. Это подтверждает нарисованная мною диаграмма.



Точка на середине отрезка, соединяющего точки (5, 3) и (−2, 1), имеет координаты (3/2, 2)

Упражнение 4.14. Вернемся к векторному преобразованию $S(\mathbf{v})$, которое возводит в квадрат все координаты, превращая $\mathbf{v} = (x, y)$ в $S(\mathbf{v}) = (x^2, y^2)$. Нарисуйте все 36 векторов **v** с целочисленными координатами от 0 до 5 в виде точек, используя функцию рисования из главы 2, а затем для каждой нарисуйте результат $S(\mathbf{v})$. Как S влияет на геометрическое расположение векторов?

Решение. Изначально расстояния между точками одинаковые, но после применения преобразования, с увеличением координат x и y, расстояние увеличивается в горизонтальном и вертикальном направлениях соответственно.



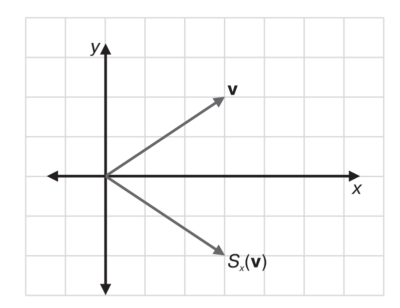
Изначально точки равномерно распределены по охватываемой области, но после преобразования S расстояния между точками меняются и по вертикали, и по горизонтали

Упражнение 4.15. Мини-проект. *Тестирование на основе свойств* — это разновидность модульного тестирования, которое включает создание произвольных входных данных для программы и последующую проверку соответствия ожиданиям результатов на выходе. Есть даже несколько популярных библиотек для Python, например Hypothesis (можно установить с помощью `pip`), поддерживающих такое тестирование. Выберите библиотеку на свой вкус и реализуйте тесты, проверяющие линейность векторного преобразования.

В частности, для векторного преобразования T, реализованного в виде функции на Python, сгенерируйте большое количество пар случайных векторов и подтвердите, что для всех них преобразование T сохраняет сумму. Затем напишите похожий тест для пар, включающих скаляр и вектор, и убедитесь, что T сохраняет произведение на скаляр. Подтвердите, что линейные преобразования, такие как `rotate_x_by(pi/2)`, успешно проходят тест, а нелинейные, такие как возведение координат в квадрат, не проходят.

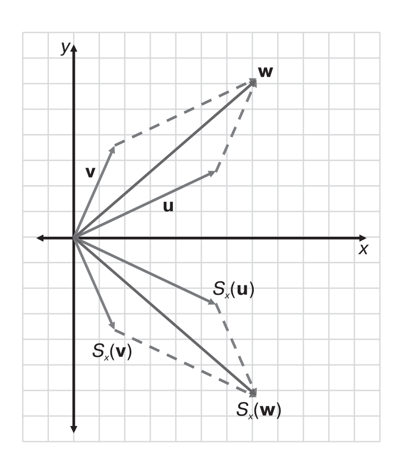
Упражнение 4.16. *Отражение* относительно оси x — одно из двухмерных векторных преобразований. Оно принимает вектор и возвращает другой вектор, который является зеркальным отражением первого относительно оси x. Их координаты x равны, а координаты y имеют разный знак, оставаясь равными по абсолютной величине. Обозначив это преобразование S_x, получаем образ вектора $\mathbf{v} = (3, 2)$ и преобразованного вектора $S_x(\mathbf{v})$.

Нарисуйте два вектора и их сумму, а также отражения этих трех векторов, чтобы продемонстрировать, что это преобразование сохраняет сумму векторов. Нарисуйте еще один график, чтобы показать, что оно также сохраняет произведение вектора на скаляр, тем самым подтвердив выполнение обоих критериев линейности.



Вектор $\mathbf{v} = (3, 2)$ и его отражение $(3, -2)$ относительно оси x

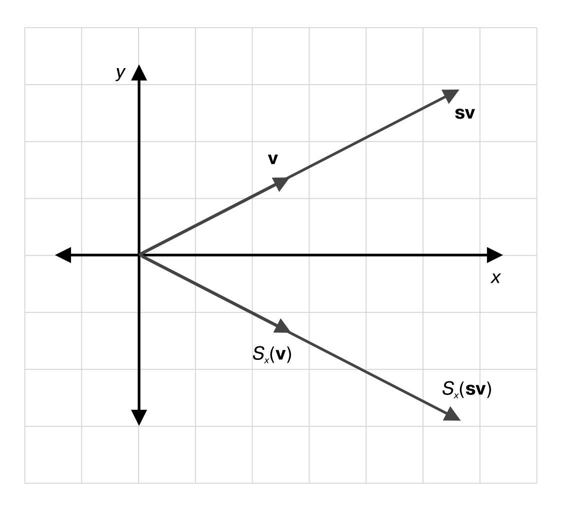
Решение. Вот пример отражения относительно оси x, подтверждающий сохранение векторной суммы.



Для $\mathbf{u} + \mathbf{v} = \mathbf{w}$ отражение относительно оси x сохраняет сумму, то есть $S_x(\mathbf{u}) + S_x(\mathbf{v}) = S_x(\mathbf{w})$

Следующий пример отражения подтверждает сохранение произведения вектора на скаляр: $S_x(s\mathbf{v})$ находится именно там, где ожидается $sS_x(\mathbf{v})$.

Чтобы *доказать*, что преобразование S_x линейное, нужно показать, что аналогичные изображения получаются для каждой векторной суммы и каждого произведения вектора на скаляр. Однако их бесконечно много, поэтому лучше использовать алгебраическое доказательство. (Сможете ли вы сами доказать эти два факта алгебраически?)



Отражение относительно оси *x* сохраняет произведение вектора на скаляр

Упражнение 4.17. Мини-проект. Пусть S и T — линейные преобразования. Объясните, почему композиция S и T также линейна.

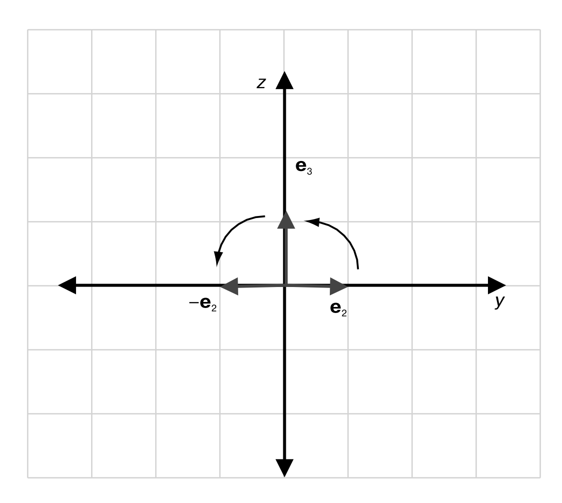
Решение. Композиция $S(T(\mathbf{v}))$ линейна, если для любой векторной суммы $\mathbf{u} + \mathbf{v} = \mathbf{w}$ выполняется равенство $S(T(\mathbf{u})) + S(T(\mathbf{v})) = S(T(\mathbf{w}))$ и для любого произведения вектора на скаляр $s\mathbf{v}$ выполняется равенство $S(T(s\mathbf{v})) = s \cdot S(T(\mathbf{v}))$. Но это лишь изложение условий, которые должны быть удовлетворены.

Теперь посмотрим, почему они истинны. Предположим сначала, что $\mathbf{u} + \mathbf{v} = \mathbf{w}$ для любых данных векторов $\mathbf{u}$ и $\mathbf{v}$. Тогда в силу линейности T выполняется равенство $T(\mathbf{u}) + T(\mathbf{v}) = T(\mathbf{w})$. Поскольку утверждается, что S — линейное преобразование, сумма должна сохраняться и для S: $S(T(\mathbf{u})) + S(T(\mathbf{v})) = S(T(\mathbf{w}))$. То есть $S(T(\mathbf{v}))$ сохраняет векторную сумму.

Точно так же для любого произведения вектора на скаляр $s\mathbf{v}$ в силу линейности T выполняется равенство $s \cdot T(\mathbf{v}) = T(s\mathbf{v})$. Также, согласно утверждению о линейности S, $s \cdot S(T(\mathbf{v})) = S(T(s\mathbf{v}))$. Это означает, что $S(T(\mathbf{v}))$ сохраняет произведение вектора на скаляр и, следовательно, что $S(T(\mathbf{v}))$ удовлетворяет полному определению линейности. Отсюда можно сделать вывод, что композиция двух линейных преобразований линейна.

Упражнение 4.18. Пусть T — линейное преобразование, реализованное как функция `rotate_x_by(pi/2)`. Что получится в результате применения преобразования $T(\mathbf{e}_1)$, $T(\mathbf{e}_2)$ и $T(\mathbf{e}_3)$?

Решение. Любой поворот вокруг оси не затрагивает точки, лежащие на этой оси, поэтому, поскольку $\mathbf{e}_1$ лежит на оси x, $T(\mathbf{e}_1) = \mathbf{e}_1 = (1, 0, 0)$. Поворот $\mathbf{e}_2 = (0, 1, 0)$ против часовой стрелки в плоскости yz перенесет этот вектор из точки, лежащей на положительной оси y на расстоянии одной единицы от начала координат, в точку, лежащую на положительной оси z также на расстоянии одной единицы от начала координат, то есть $T(\mathbf{e}_2) = \mathbf{e}_3 = (0, 0, 1)$. Аналогично, поворот $\mathbf{e}_3$ против часовой стрелки перенесет этот вектор из точки, лежащей на положительной оси z, в точку на отрицательной оси y. $T(\mathbf{e}_3)$ по-прежнему будет иметь длину, равную единице, соответственно, это будет вектор $-\mathbf{e}_2$.



Четверть оборота против часовой стрелки в плоскости yz перенесет $\mathbf{e}_2$ в $\mathbf{e}_3$ и $\mathbf{e}_3$ в $-\mathbf{e}_2$

Упражнение 4.19. Напишите функцию `linear_combination(scalars, *vectors)`, которая принимает список скаляров и такое же количество векторов и возвращает один вектор. Например, вызов `linear_combination([1,2,3], (1,0,0), (0,1,0), (0,0,1))` должен вернуть $1 \cdot (1, 0, 0) + 2 \cdot (0, 1, 0) + 3 \cdot (0, 0, 1)$ или $(1, 2, 3)$.

Решение

```
from vectors import *
def linear_combination(scalars,*vectors):
    scaled = [scale(s,v) for s,v in zip(scalars,vectors)]
    return add(*scaled)
```

Проверим, дает ли эта функция ожидаемый результат, как было показано ранее:

```
>>> linear_combination([1,2,3], (1,0,0), (0,1,0), (0,0,1))
(1, 2, 3)
```

Упражнение 4.20. Напишите функцию `transform_standard_basis(transform)`, которая принимает трехмерное векторное преобразование и возвращает результат его применения к стандартному базису. Функция должна возвращать кортеж с тремя векторами — результатом применения преобразования к $\mathbf{e}_1$, $\mathbf{e}_2$ и $\mathbf{e}_3$.

Решение. Чтобы получить желаемый результат, достаточно применить преобразование к каждому вектору стандартного базиса:

```
def transform_standard_basis(transform):
    return transform((1,0,0)), transform((0,1,0)), transform((0,0,1))
```

Правильность вычислений подтверждается (в пределах ошибки вычислений с плавающей точкой) решением упражнения 4.18, где мы определяли результат `rotate_x_by(pi/2)`:

```
>>> from math import *
>>> transform_standard_basis(rotate_x_by(pi/2))
((1, 0.0, 0.0), (0, 6.123233995736766e-17, 1.0), (0, -1.0,
    1.2246467991473532e-16))
```

Эти векторы приблизительно соответствуют векторам $(1, 0, 0)$, $(0, 0, 1)$ и $(0, -1, 0)$.

Упражнение 4.21. Пусть B — линейное преобразование такое, что $B(\mathbf{e}_1) = (0, 0, 1)$, $B(\mathbf{e}_2) = (2, 1, 0)$, $B(\mathbf{e}_3) = (-1, 0, -1)$ и $\mathbf{v} = (-1, 1, 2)$. Определите результат $B(\mathbf{v})$.

Решение. Согласно условию $\mathbf{v} = (-1, 1, 2) = -\mathbf{e}_1 + \mathbf{e}_2 + 2\mathbf{e}_3$, $B(\mathbf{v}) = B(-\mathbf{e}_1 + \mathbf{e}_2 + 2\mathbf{e}_3)$. Поскольку B — линейное преобразование, то оно должно сохранять линейную комбинацию $B(\mathbf{v}) = -B(\mathbf{e}_1) + B(\mathbf{e}_2) + 2 \cdot B(\mathbf{e}_3)$. Отсюда следует: $B(\mathbf{v}) = -(0, 0, 1) + (2, 1, 0) + 2 \cdot (-1, 0, -1) = (0, 1, -3)$.

Упражнение 4.22. Пусть A и B — линейные преобразования такие, что $A(\mathbf{e}_1) = (1, 1, 1)$, $A(\mathbf{e}_2) = (1, 0, -1)$, $A(\mathbf{e}_3) = (0, 1, 1)$, $B(\mathbf{e}_1) = (0, 0, 1)$, $B(\mathbf{e}_2) = (2, 1, 0)$ и $B(\mathbf{e}_3) = (-1, 0, -1)$. Определите результаты $A(B(\mathbf{e}_1))$, $A(B(\mathbf{e}_2))$ и $A(B(\mathbf{e}_3))$.

Решение. $A(B(\mathbf{e}_1))$ — это применение преобразования A к $B(\mathbf{e}_1) = (0, 0, 1) = \mathbf{e}_3$. Мы уже знаем, что $A(\mathbf{e}_3) = (0, 1, 1)$, поэтому $A(B(\mathbf{e}_1)) = (0, 1, 1)$.

$A(B(\mathbf{e}_2))$ — это применение преобразования A к $B(\mathbf{e}_2) = (2, 1, 0)$. Это линейная комбинация $A(\mathbf{e}_1)$, $A(\mathbf{e}_2)$ и $A(\mathbf{e}_3)$ со скалярами $(2, 1, 0)$: $2 \cdot (1, 1, 1) + 1 \cdot (1, 0, -1) + 0 \cdot (0, 1, 1) = (3, 2, 1)$.

Наконец, $A(B(\mathbf{e}_3))$ — это применение преобразования A к $B(\mathbf{e}_3) = (-1, 0, -1)$. Это линейная комбинация $-1 \cdot (1, 1, 1) + 0 \cdot (1, 0, -1) + -1 \cdot (0, 1, 1) = (-1, -2, -2)$.

Обратите внимание на то, что теперь мы знаем результат композиции A и B для всех векторов стандартного базиса, поэтому легко можем вычислить $A(B(\mathbf{v}))$ для любого вектора $\mathbf{v}$.

Линейные преобразования хорошо зарекомендовали себя и легко вычисляются, потому что задаются небольшим объемом данных. Мы подробнее поговорим об этом в следующей главе, когда будем вычислять линейные преобразования с использованием *матричных* обозначений.

КРАТКИЕ ИТОГИ ГЛАВЫ

- Векторные преобразования — это функции, которые принимают и возвращают векторы. Векторные преобразования могут работать с двух- и трехмерными векторами.

- Чтобы выполнить геометрическое преобразование модели, нужно применить векторное преобразование к каждой вершине каждого многоугольника трехмерной модели.
- Существующие векторные преобразования можно комбинировать и тем самым создавать новые преобразования, эквивалентные последовательному применению существующих векторных преобразований.
- Функциональное программирование — это парадигма программирования, в которой особое внимание уделяется созданию функций и управлению ими.
- Функциональная операция каррирования превращает функцию, которая принимает несколько аргументов, в функцию, которая принимает один аргумент и возвращает новую функцию. Каррирование позволяет превращать существующие функции, такие как `scale` и `add`, в векторные преобразования.
- Линейные преобразования — это векторные преобразования, сохраняющие векторные суммы и произведения векторов на скаляры. В частности, точки, лежащие на отрезке, после применения линейного преобразования остаются лежать на нем.
- Линейная комбинация — это наиболее общая комбинация произведения вектора на скаляр и векторной суммы. Каждый трехмерный вектор фактически является линейной комбинацией векторов трехмерного стандартного базиса, которые обозначаются $\mathbf{e}_1 = (1, 0, 0)$, $\mathbf{e}_2 = (0, 1, 0)$ и $\mathbf{e}_3 = (0, 0, 1)$. Точно так же каждый двухмерный вектор — это линейная комбинация векторов двухмерного стандартного базиса $\mathbf{e}_1 = (1, 0)$ и $\mathbf{e}_2 = (0, 1)$.
- Зная, как данное линейное преобразование действует на векторы стандартного базиса, легко можно определить, как оно действует на любой вектор, для чего достаточно записать последний как линейную комбинацию стандартного базиса и использовать тот факт, что линейные комбинации сохраняют свойства отношений.
  - В трехмерном пространстве линейное преобразование задается тремя векторами или девятью числами.
  - В двухмерном пространстве линейное преобразование задается двумя векторами или четырьмя числами.

  Этот пункт имеет основополагающее значение: линейные преобразования хорошо зарекомендовали себя и легко вычисляются, потому что задаются небольшим объемом данных.

5

Вычисление преобразований с помощью матриц

В этой главе

- ✓ Запись линейного преобразования в виде матрицы.
- ✓ Умножение матриц для объединения и применения линейных преобразований.
- ✓ Операции с векторами разной мерности с использованием линейных преобразований.
- ✓ Параллельный перенос двух- или трехмерных векторов с помощью матриц.

В главе 4 я сформулировал важное утверждение: любое линейное преобразование в трехмерном пространстве можно задать всего тремя векторами, или девятью числами. Правильно подобрав эти девять чисел, можно выполнить поворот на любой угол вокруг любой оси, отражение относительно любой плоскости, проекцию на любую плоскость, масштабирование с любым коэффициентом в любом направлении и любое другое линейное трехмерное преобразование.

Преобразование, выраженное словами «поворот против часовой стрелки на 90° вокруг оси z», можно эквивалентно описать в виде действия с векторами стандартного базиса $\mathbf{e}_1 = (1, 0, 0)$, $\mathbf{e}_2 = (0, 1, 0)$ и $\mathbf{e}_3 = (0, 0, 1)$, результаты которого в данном случае равны $(0, 1, 0)$, $(-1, 0, 0)$ и $(0, 0, 1)$. Представляя это преобразование геометрически или описывая его тремя векторами (или девятью числами),

мы рассуждаем об одной и той же воображаемой машине (рис. 5.1), которая работает с трехмерными векторами. Конструкции машин могут быть разными, но все они дают одинаковые результаты.

Таблица с числами, описывающими линейное преобразование, называется *матрицей*. В этой главе основное внимание уделяется использованию таких сеток чисел в качестве вычислительных инструментов, поэтому здесь мы будем работать с числами больше, чем в предыдущих главах. Но пусть вас это не пугает! В конце концов мы все еще имеем дело с простыми векторными преобразованиями.

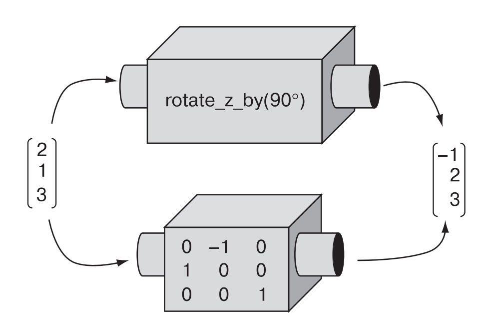


Рис. 5.1. Две машины, выполняющие одно и то же линейное преобразование. Геометрические рассуждения приводят в действие машину вверху, а девять чисел — машину внизу

Матрица позволяет вычислить заданное линейное преобразование, предоставляя данные о влиянии этого преобразования на векторы стандартного базиса. Все обозначения в этой главе служат для описания процесса, рассмотренного в разделе 4.2, а не для введения новых и незнакомых тем. Я знаю, что изучение новых и сложных обозначений бывает непростым, но обещаю, что эта наука пойдет вам на пользу. Векторы лучше всего представлять себе как геометрические объекты или как кортежи чисел. Точно так же линейные преобразования лучше представлять как числовые матрицы.

5.1. ПРЕДСТАВЛЕНИЕ ЛИНЕЙНЫХ ПРЕОБРАЗОВАНИЙ В ВИДЕ МАТРИЦ

Вернемся к конкретному примеру девяти чисел, определяющих трехмерное линейное преобразование. Предположим, что A — это линейное преобразование, о котором известно, что $A(\mathbf{e}_1) = (1, 1, 1)$, $A(\mathbf{e}_2) = (1, 0, -1)$ и $A(\mathbf{e}_3) = (0, 1, 1)$. Эти три вектора, включающие в общей сложности девять компонентов, содержат всю информацию, определяющую линейное преобразование A.

Поскольку мы будем использовать это понятие снова и снова, введем для него специальное обозначение. Возьмем на вооружение новую форму записи, называемую *матричной записью*, для работы с этими девятью числами, представляющими A.

5.1.1. Запись векторов и линейных преобразований в виде матриц

Матрицы — это прямоугольные таблицы чисел, и их форма говорит нам, как их интерпретировать. Например, матрицу, состоящую из одного столбца, можно интерпретировать как вектор, элементы которого представляют координаты, упорядоченные сверху вниз. Векторы этого вида называются *векторами-столбцами*. Например, стандартный базис трехмерного пространства можно записать в виде трех векторов-столбцов:

$$\mathbf{e}_1 = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix};\ \mathbf{e}_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix};\ \mathbf{e}_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

Для нас эта запись означает то же самое, что и $\mathbf{e}_1 = (1, 0, 0)$, $\mathbf{e}_2 = (0, 1, 0)$ и $\mathbf{e}_3 = (0, 0, 1)$. Аналогичным образом можно указать, как A преобразует векторы стандартного базиса:

$$A(\mathbf{e}_1) = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix};\ A(\mathbf{e}_2) = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix};\ A(\mathbf{e}_3) = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}.$$

Матрица, представляющая линейное преобразование A, имеет вид таблицы 3×3, объединяющей эти векторы:

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & -1 & 1 \end{pmatrix}.$$

Двухмерный вектор-столбец состоит из двух элементов, поэтому два вектора, описывающих преобразование, всего содержат четыре элемента. Рассмотрим линейное преобразование D, масштабирующее входные векторы, умножая их на 2. Сначала запишем результат преобразования векторов базиса:

$$D(\mathbf{e}_1) = \begin{pmatrix} 2 \\ 0 \end{pmatrix};\ D(\mathbf{e}_2) = \begin{pmatrix} 0 \\ 2 \end{pmatrix},$$

затем — матрицу, описывающую D, поместив эти столбцы рядом друг с другом:

$$D = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}.$$

Матрицы могут быть других форм и размеров, но сейчас мы сосредоточимся на этих двух: матрицах из одного столбца, представляющих векторы, и квадратных матрицах, представляющих линейные преобразования.

Повторю еще раз, что здесь нет никаких новых концепций — просто другой способ записи основной темы раздела 4.2: линейное преобразование определяется результатами воздействия на векторы стандартного базиса. Чтобы получить матрицу линейного преобразования, нужно найти векторы, полученные применением этого преобразования ко всем векторам стандартного базиса, и объединить результаты в одну таблицу. Теперь мы рассмотрим противоположную задачу: как вычислить результат линейного преобразования по его матрице.

5.1.2. Умножение матрицы на вектор

Если линейное преобразование B представлено матрицей и вектор $\mathbf{v}$ тоже представлен матрицей (вектором-столбцом), то можно вычислить результат $B(\mathbf{v})$. Например, если B и $\mathbf{v}$ заданы так:

$$B = \begin{pmatrix} 0 & 2 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix};\ \mathbf{v} = \begin{pmatrix} 3 \\ -2 \\ 5 \end{pmatrix},$$

то векторы $B(\mathbf{e}_1)$, $B(\mathbf{e}_2)$ и $B(\mathbf{e}_3)$ можно получить из B как столбцы матрицы. Теперь используем ту же процедуру, что и раньше. Поскольку $\mathbf{v} = 3\mathbf{e}_1 - 2\mathbf{e}_2 + 5\mathbf{e}_3$, отсюда следует, что $B(\mathbf{v}) = 3B(\mathbf{e}_1) - 2B(\mathbf{e}_2) + 5B(\mathbf{e}_3)$. Развернув это выражение, получаем:

$$B(\mathbf{v}) = 3 \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} - 2 \cdot \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix} + 5 \cdot \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 3 \end{pmatrix} + \begin{pmatrix} -4 \\ -2 \\ 0 \end{pmatrix} + \begin{pmatrix} 5 \\ 0 \\ -5 \end{pmatrix} = \begin{pmatrix} 1 \\ -2 \\ -2 \end{pmatrix},$$

то есть результатом является вектор (1, –2, –2). Интерпретация квадратной матрицы как функции, оперирующей вектором-столбцом, — это частный случай операции, называемой *матричным умножением*. И снова мы сталкиваемся с новыми обозначениями и терминологией, но в действительности делаем все то же самое — применяем линейное преобразование к вектору. Вот как выглядит линейное преобразование, записанное в виде матричного умножения:

$$B\mathbf{v} = \begin{pmatrix} 0 & 2 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} 3 \\ -2 \\ 5 \end{pmatrix} = \begin{pmatrix} 1 \\ -2 \\ -2 \end{pmatrix}.$$

В отличие от умножения чисел, в матричном умножении порядок имеет значение. В этом случае $B\mathbf{v}$ — это допустимое умножение, а $\mathbf{v}B$ — нет. Вскоре вы увидите, как перемножать матрицы разной формы, и узнаете общее правило

перемножения матриц. А пока поверьте мне на слово и считайте это умножение допустимым, потому что оно означает применение трехмерного линейного оператора к трехмерному вектору.

Умножение матриц легко реализовать на Python. Представим матрицу B как кортеж кортежей, а вектор **v** — как кортеж:

```
B = (
    (0,2,1),
    (0,1,0),
    (1,0,-1)
)

v = (3,-2,5)
```

Это представление немного отличается от первоначального представления матрицы B как объединения трех столбцов: здесь B описывается как последовательность строк. Преимущество определения матрицы в виде кортежа строк заключается в том, что числа располагаются в том же порядке, в каком мы записываем их на бумаге. Однако мы в любой момент можем получить столбцы, вызвав функцию `zip`, описанную в приложении Б:

```
>>> list(zip(*B))
[(0, 0, 1), (2, 1, 0), (1, 0, -1)]
```

Первый элемент этого списка — (0, 0, 1) — это первый столбец матрицы B и т. д. Далее нужно получить линейную комбинацию этих векторов, в которой скалярами выступают координаты **v**. Для этого можно использовать функцию `linear_combination` из упражнения 4.19, приведенного в подразделе 4.2.5. Первым аргументом в вызов `linear_combination` нужно передать **v**, служащий списком скаляров, а последующие аргументы должны быть столбцами B. Вот как выглядит законченная функция:

```
def multiply_matrix_vector(matrix, vector):
    return linear_combination(vector, *zip(*matrix))
```

Ее правильность подтверждается вычислениями, которые мы выполнили с B и **v** вручную:

```
>>> multiply_matrix_vector(B,v)
(1, -2, -2)
```

Есть еще два рецепта, помогающих запомнить порядок умножения матрицы на вектор, оба дают одинаковый результат. Для знакомства с ними запишем прототип матричного умножения:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}\begin{pmatrix} x \\ y \\ z \end{pmatrix}.$$

Результат — линейная комбинация столбцов матрицы и координат x, y и z, выступающих в роли скаляров:

$$= x \cdot \begin{pmatrix} a \\ d \\ g \end{pmatrix} + y \cdot \begin{pmatrix} b \\ e \\ h \end{pmatrix} + z \cdot \begin{pmatrix} c \\ f \\ i \end{pmatrix} = \begin{pmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{pmatrix}.$$

Это явная формула произведения матрицы 3×3 на трехмерный вектор. Аналогично можно записать умножение матрицы 2×2 на двухмерный вектор:

$$\begin{pmatrix} j & k \\ l & m \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = x \cdot \begin{pmatrix} j \\ l \end{pmatrix} + y \cdot \begin{pmatrix} k \\ m \end{pmatrix} = \begin{pmatrix} jx + ky \\ lx + my \end{pmatrix}.$$

Первое правило гласит: каждая координата выходного вектора является функцией всех координат входного вектора. Например, первая координата трехмерного выходного вектора — это функция $f(x, y, z) = ax + by + cz$. Кроме того, данная функция линейная в том смысле, в каком нам рассказывали на уроках алгебры в школе, — это сумма произведений всех переменных. Мы сначала ввели термин «линейное преобразование», потому что линейные преобразования сохраняют линии. Однако линейное преобразование — это еще и набор линейных *функций* входных координат, которые дают соответствующие выходные координаты.

Второе правило гласит: координаты выходного вектора являются скалярными произведениями строк матрицы на заданный вектор. Например, умножение первой строки матрицы 3×3 (a, b, c) на вектор (x, y, z) дает первую координату выходного вектора: $(a, b, c) \cdot (x, y, z) = ax + by + cz$. Эти две формы записи можно объединить в формулу:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} (a, b, c) \cdot (x, y, z) \\ (d, e, f) \cdot (x, y, z) \\ (g, h, i) \cdot (x, y, z) \end{pmatrix} = \begin{pmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{pmatrix}.$$

Если у вас начинает рябить в глазах от такого обилия букв и цифр в массивах, то не волнуйтесь. Поначалу обозначения могут ошеломить, и нужно какое-то время, чтобы в них разобраться. В этом вам помогут другие примеры матриц в текущей главе и еще большее количество примеров в следующей.

5.1.3. Объединение линейных преобразований путем умножения матриц

До сих пор мы видели лишь несколько примеров линейных преобразований — повороты, отражения, масштабирование и другие геометрические преобразования. Кроме того, любое количество последовательно применяемых линейных преобразований дает новое линейное преобразование. В математической терминологии *композиция* (или объединение) любого количества линейных преобразований также является линейным преобразованием.

Поскольку любое линейное преобразование можно представить матрицей, любое объединение двух линейных преобразований также можно представить матрицей. На самом деле матрицы — лучший инструмент для объединения линейных преобразований.

ПРИМЕЧАНИЕ

Позвольте мне ненадолго снять колпак математика и надеть колпак программиста. Допустим, нам нужно вычислить результат применения к вектору последовательности, скажем, из 1000 линейных преобразований. Это может потребоваться, например, для воспроизведения анимационного эффекта с участием некоторого объекта путем применения небольших преобразований в каждом кадре. Последовательное применение 1000 функций на языке Python потребовало бы выполнения очень большого объема вычислений. Однако, если найти матрицу, представляющую композицию из 1000 линейных преобразований, то можно свести весь процесс к горстке чисел и вычислений.

Рассмотрим композицию двух линейных преобразований $A(B(\mathbf{v}))$, где известно, что A и B имеют следующие матричные представления:

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & -1 & 1 \end{pmatrix};\ B = \begin{pmatrix} 0 & 2 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix}.$$

Вот как реализуется эта композиция. Сначала к вектору $\mathbf{v}$ применяется преобразование B, что дает новый вектор $B(\mathbf{v})$ или $B\mathbf{v}$, если записать применение как умножение. Далее к этому вектору применяется преобразование A, и в результате получается итоговый трехмерный вектор $A(B\mathbf{v})$. Снова опустим скобки и запишем $A(B\mathbf{v})$ как произведение $AB\mathbf{v}$. Подставив в это произведение вектор $\mathbf{v} = (x, y, z)$, получим следующую формулу:

$$AB\mathbf{v} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 2 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}.$$

Как вы помните, эти вычисления выполняются справа налево. А теперь я заявляю, что их можно выполнить слева направо и результат от этого не изменится. В частности, мы можем сначала вычислить произведение матриц AB — это будет новая матрица, представляющая композицию линейных преобразований A и B:

$$AB = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 2 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{pmatrix}.$$

Но как вычислить элементы этой новой матрицы? Она должна представлять композицию преобразований A и B, дающую новое линейное преобразование AB. Как мы видели, столбцы матрицы являются результатом применения преобразования к векторам стандартного базиса. Столбцы матрицы AB являются результатом применения преобразования AB к каждому из векторов $\mathbf{e}_1$, $\mathbf{e}_2$ и $\mathbf{e}_3$.

Таким образом, столбцы матрицы AB — это $AB(\mathbf{e}_1)$, $AB(\mathbf{e}_2)$ и $AB(\mathbf{e}_3)$. Посмотрим, например, на первый столбец, который должен быть вектором $AB(\mathbf{e}_1)$, то есть результатом применения A к вектору $B(\mathbf{e}_1)$. Другими словами, чтобы получить первый столбец произведения AB, нужно умножить матрицу на вектор, а эту операцию мы с вами уже выполняли:

$$AB = \underset{A}{\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & -1 & 1 \end{pmatrix}} \underset{B(\mathbf{e}_1)}{\begin{pmatrix} 0 & 2 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix}} = \underset{AB(\mathbf{e}_1)}{\begin{pmatrix} 0 & ? & ? \\ 1 & ? & ? \\ 1 & ? & ? \end{pmatrix}}.$$

Аналогично находим $AB(\mathbf{e}_2) = (3, 2, 1)$ и $AB(\mathbf{e}_3) = (1, 0, 0)$ — второй и третий столбцы матрицы AB:

$$AB = \begin{pmatrix} 0 & 3 & 1 \\ 1 & 2 & 0 \\ 1 & 1 & 0 \end{pmatrix}.$$

Так выполняется умножение матриц. Как видите, оно заключается в простом комбинировании линейных операторов. Чтобы лучше запомнить процесс умножения, можно использовать простое рассуждение. Поскольку умножение матрицы 3×3 на вектор-столбец аналогично трем скалярным произведениям, умножение двух матриц 3×3 равноценно девяти скалярным произведениям — всех возможных скалярных произведений строк первой матрицы на столбцы второй (рис. 5.2).

$$B = \begin{pmatrix} 0 & 2 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix}$$

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & -1 & 1 \end{pmatrix} \quad \begin{pmatrix} 0 & 3 & 1 \\ 1 & 2 & 0 \\ 1 & 1 & 0 \end{pmatrix} = AB$$

$$(1,0,1) \cdot (2,1,0) = 1 \cdot 2 + 0 \cdot 1 + 1 \cdot 0 = 2$$

Рис. 5.2. Каждый элемент матрицы результата — это скалярное произведение строки первой матрицы на столбец второй

Все, что говорилось об умножении матриц 3×3, применимо и к матрицам 2×2. Например, чтобы найти произведение матриц 2×2

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix},$$

нужно найти скалярные произведения векторов-строк первой матрицы на векторы-столбцы второй. Скалярное произведение первой строки в матрице слева на первый столбец в матрице справа равно $(1, 2) \cdot (0, 1) = 2$. Соответственно, элемент, принадлежащий первой строке и первому столбцу матрицы результата, равен 2:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & ? \\ ? & ? \end{pmatrix}.$$

Повторив процедуру, находим оставшиеся элементы матричного произведения:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}=\begin{pmatrix} 2 & -1 \\ 4 & 3 \end{pmatrix}.$$

Вы можете попрактиковаться и перемножить несколько пар матриц вручную, но в будущем почти наверняка предпочтете переложить эту работу на компьютер. Реализуем матричное умножение на Python, чтобы обеспечить себе такую возможность.

5.1.4. Реализация умножения матриц

Реализовать умножение матриц можно несколькими способами, я предпочитаю использовать трюк со скалярным произведением. Поскольку результатом умножения матриц должен быть кортеж кортежей, можно записать умножение как вложенные друг в друга генераторы списков. Функция принимает два кортежа кортежей, `a` и `b`, представляющих исходные матрицы A и B. Матрица `a` уже является кортежем строк первой матрицы, и с ней не нужно ничего делать, а вот матрицу `b` следует преобразовать в кортеж столбцов, что можно сделать вызовом `zip(*b)`. Наконец, во внутреннем генераторе нужно найти скалярное произведение для каждой пары векторов и передать его внешнему генератору:

```
from vectors import *

def matrix_multiply(a,b):
    return tuple(
        tuple(dot(row,col) for col in zip(*b))
        for row in a
    )
```

Внешний генератор создает строки результата, а внутренний — отдельные элементы строк. Поскольку выходные строки формируются различными скалярными произведениями со строками `a`, внешний генератор выполняет итерации по `a`.

Функция `matrix_multiply` не имеет жестко ограниченного числа измерений, поэтому ее можно использовать для перемножения двух- и трехмерных матриц. Перемножим для пробы матрицы из предыдущих примеров:

```
>>> a = ((1,1,0),(1,0,1),(1,-1,1))
>>> b = ((0,2,1),(0,1,0),(1,0,-1))
>>> matrix_multiply(a,b)
((0, 3, 1), (1, 2, 0), (1, 1, 0))
>>> c = ((1,2),(3,4))
>>> d = ((0,-1),(1,0))
>>> matrix_multiply(c,d)
((2, -1), (4, -3))
```

Имея вычислительный инструмент умножения матриц, теперь мы можем выполнять некоторые простые манипуляции с трехмерной графикой.

5.1.5. Анимация в трехмерном пространстве с помощью матричных преобразований

Чтобы создать анимационный эффект, в каждом кадре нужно перерисовать трансформированную версию исходной трехмерной модели. Чтобы модель выглядела движущейся или меняющейся с течением времени, нужно в разные моменты применять разные преобразования. Если преобразования являются линейными и задаются матрицами, то для создания каждого нового кадра анимации нужна новая матрица.

Поскольку в PyGame имеются встроенные часы, способные отслеживать течение времени с точностью до миллисекунды, мы можем генерировать матрицы через равные интервалы времени. Иначе говоря, можем думать о матрице как о функции, которая принимает текущее время t и возвращает число (рис. 5.3).

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \to \begin{pmatrix} a(t) & b(t) & c(t) \\ d(t) & e(t) & f(t) \\ g(t) & h(t) & i(t) \end{pmatrix}$$

Рис. 5.3. Представление элементов матрицы как функции от времени позволяет изменять матрицу целиком с течением времени

Например, можно использовать следующие девять выражений:

$$\begin{pmatrix} \cos(t) & 0 & -\sin(t) \\ 0 & 1 & 0 \\ \sin(t) & 0 & \cos(t) \end{pmatrix}.$$

Как рассказывалось в главе 2, косинус и синус — это функции, которые принимают и возвращают число. Остальные пять элементов не меняются со временем, но для единообразия мысленной модели их можно представлять как константные функции (например, $f(t) = 1$ в центральном элементе). При любом значении t эта матрица представляет то же линейное преобразование, что и `rotate_y_by(t)`. Время движется вперед, и значение t увеличивается, поэтому, если применить это матричное преобразование к каждому кадру, мы каждый раз будем получать поворот на больший угол.

Передадим функции `draw_model`, описанной в приложении В и широко используемой в главе 4, именованный аргумент `get_matrix` с функцией, которая сама получает время в миллисекундах и возвращает матрицу преобразования, соответствующую этому моменту времени. Я вызываю ее в файле `animate_teapot.py`, чтобы воспроизвести эффект поворота чайника из главы 4:

```
from teapot import load_triangles
from draw_model import draw_model
from math import sin,cos
```

```
def get_rotation_matrix(t):
    seconds = t/1000
    return (
        (cos(seconds),0,-sin(seconds)),
        (0,1,0),
        (sin(seconds),0,cos(seconds))
    )

draw_model(load_triangles(),
           get_matrix=get_rotation_matrix)
```

Создает новую матрицу преобразования, соответствующую входному числу, представляющему время

Перевести время в секунды, чтобы преобразования не происходили слишком быстро

Передать в вызов draw_model именованный аргумент с функцией

Теперь в `draw_model` передаются данные, необходимые для преобразования базовой модели чайника с течением времени, и их нужно как-то использовать в теле функции. Прежде чем перебирать грани чайника, применим к модели соответствующее матричное преобразование:

```
def draw_model(faces, color_map=blues, light=(1,2,3),
               camera=Camera("default_camera",[]),
               glRotatefArgs=None,
               get_matrix=None):
    #...
    def do_matrix_transform(v):
        if get_matrix:
            m = get_matrix(pygame.time.get_ticks())
            return multiply_matrix_vector(m, v)
        else:
            return v
    transformed_faces = polygon_map(do_matrix_transform,
                                    faces)
    for face in transformed_faces:
    #...
```

Большая часть тела функции осталась неизменной, поэтому она здесь не приводится

Новая функция в теле главного цикла while, которая применяет матричное преобразование, соответствующее текущему кадру

Если в аргументе get_matrix ничего не было передано, то никаких преобразований не выполняется и вектор возвращается без изменений

Применить функцию к каждому многоугольнику с помощью polygon_map

Остальная часть функции draw_model не изменилась, как показано в приложении B

Тело этого оператора использует прошедшее время в миллисекундах, возвращаемое вызовом pygame.time.get_ticks(), а также функцию, переданную в get_matrix, для вычисления матрицы преобразования для этого кадра

Добавив эти изменения, запустите сценарий, и вы увидите поворачивающийся чайник (рис. 5.4).



Рис. 5.4. В каждом кадре чайник преобразуется новой матрицей, соответствующей времени, прошедшему с начала воспроизведения анимации

Надеюсь, приведенные примеры убедили вас, что матрицы полностью взаимозаменяемы с линейными преобразованиями. Мы смогли воспроизвести анимационный эффект с чайником, описывая преобразования всего девятью числами. Теперь можете усовершенствовать навыки работы с матрицами в следующих упражнениях, а затем я покажу вам, как еще можно использовать реализованную нами функцию `matrix_multiply`.

5.1.6. Упражнения

Упражнение 5.1. Напишите функцию `infer_matrix(n, transformation)`, которая принимает количество измерений (например, 2 или 3), и функцию, представляющую линейное векторное преобразование. Она должна вернуть квадратную матрицу $n \times n$ (кортеж с n кортежами по n чисел в каждом, представляющий матрицу линейного преобразования). Конечно, результат имеет смысл, только если входное преобразование линейно, иначе полученная матрица будет представлять совершенно другую функцию!

Решение

```
def infer_matrix(n, transformation):
    def standard_basis_vector(i):
        return tuple(1 if i==j else 0 for j in range(1,n+1))
    standard_basis = [standard_basis_vector(i) for i in range(1,n+1)]
    cols = [transformation(v) for v in standard_basis]
    return tuple(zip(*cols))
```

Создать i-й вектор стандартного базиса как кортеж, содержащий единицу в i-й координате и нули во всех остальных

Создать стандартный базис как список векторов

Определить столбцы матрицы как результат применения соответствующего линейного преобразования к векторам стандартного базиса

Преобразовать матрицу из кортежа столбцов в кортеж строк, чтобы соответствовать принятым нами соглашениям

Эту функцию можно проверить на примере известного преобразования, такого как `rotate_z_by(pi/2)`:

```
>>> from transforms import rotate_z_by
>>> from math import pi
>>> infer_matrix(3,rotate_z_by(pi/2))
((6.123233995736766e-17, -1.0, 0.0), (1.0, 1.2246467991473532e-16, 0.0),
(0, 0, 1))
```

Упражнение 5.2. Вычислите результат следующего произведения матрицы 2×2 на двухмерный вектор:

$$\begin{pmatrix} 1,3 & 0,7 \\ 6,5 & 3,2 \end{pmatrix} \begin{pmatrix} -2,5 \\ 0,3 \end{pmatrix}.$$

Решение. Скалярное произведение первой строки матрицы на вектор $-2,5 \cdot 1,3 + 0,3 \cdot (-0,7) = -3,46$. Скалярное произведение второй строки матрицы на вектор $-2,5 \cdot 6,5 + 0,3 \cdot 3,2 = -15,29$. Эти числа — координаты выходного вектора, то есть результатом будет вектор

$$\begin{pmatrix} 1,3 & 0,7 \\ 6,5 & 3,2 \end{pmatrix} \begin{pmatrix} -2,5 \\ 0,3 \end{pmatrix} = \begin{pmatrix} -3,46 \\ -15,29 \end{pmatrix}.$$

Упражнение 5.3. Мини-проект. Напишите функцию `random_matrix`, которая генерирует матрицы заданного размера со случайными целыми числами. Сгенерируйте с помощью этой функции пять пар матриц 3×3. Перемножьте каждую пару вручную (для практики), а затем проверьте полученный результат с помощью функции `matrix_multiply`.

Решение. Функция `random_matrix` будет принимать параметры, определяющие количество строк, количество столбцов, а также минимальное и максимальное значения элементов:

```
from random import randint
def random_matrix(rows,cols,min=-2,max=2):
    return tuple(
        tuple(
        randint(min,max) for j in range(0,cols))
        for i in range(0,rows)
    )
```

Теперь для проверки сгенерируем матрицу 3×3 со случайными значениями элементов в диапазоне от 0 до 10:

```
>>> random_matrix(3,3,0,10)
((3, 4, 9), (7, 10, 2), (0, 7, 4))
```

Упражнение 5.4. Перемножьте пары матриц из предыдущего упражнения в обратном порядке. Получится ли тот же результат?

Решение. Все результаты будут различаться, разве что вам очень повезет. Большинство пар матриц дают разные результаты при умножении в разном порядке. Операцию, результат которой не зависит от порядка операндов, на языке математики называют *коммутативной*. Например, умножение чисел — коммутативная операция, потому что $xy = yx$ для любой пары чисел x и y. Однако умножение матриц некоммутативно, потому что для двух квадратных матриц A и B результат AB не всегда равен BA.

Упражнение 5.5. И в двух-, и в трехмерном пространстве имеется малоприметное, но важное векторное преобразование, называемое *тождественным преобразованием*, которое принимает вектор и возвращает его же на выходе. Это линейное преобразование, потому что оно принимает любую входную векторную сумму, произведение вектора на скаляр или линейную комбинацию и возвращает на выходе то же самое. Как выглядят матрицы, представляющие тождественные преобразования в двух- и трехмерном пространствах?

Решение. В двух- или трехмерном пространстве тождественное преобразование оставляет неизменными векторы стандартного базиса. Следовательно, матрица тождественного преобразования для пространства любой мерности состоит из столбцов, представляющих векторы стандартного базиса. В двух- и трехмерном пространствах эти матрицы *тождественных преобразований*, или *единичные матрицы*, обозначаются I_2 и I_3 соответственно и выглядят, как показано далее:

$$I_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix};\ I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Упражнение 5.6. Примените матрицу ((2, 1, 1), (1, 2, 1), (1, 1, 2)) ко всем векторам, определяющим чайник. Что получится в результате и почему?

Решение. В исходный файл `matrix_transform_teapot.py` включена следующая функция:

```
def transform(v):
    m = ((2,1,1),(1,2,1),(1,1,2))
    return multiply_matrix_vector(m,v)

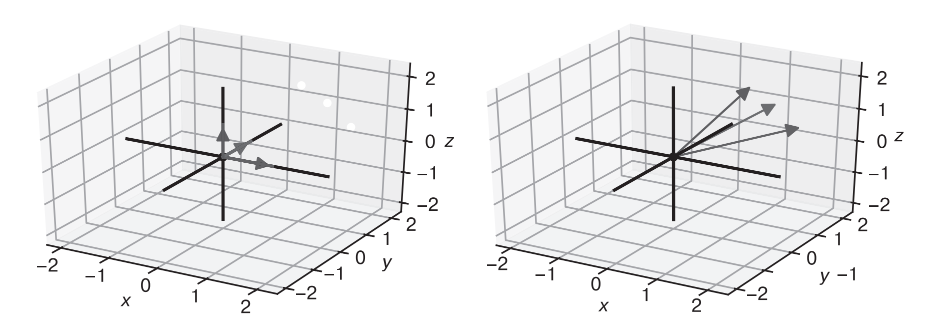
draw_model(polygon_map(transform, load_triangles()))
```

Выполнив этот код, можно увидеть, что передняя часть чайника вытянулась в положительных направлениях осей x, y и z.



Применение заданной матрицы ко всем вершинам многоугольников, составляющих чайник

Это связано с тем, что все векторы стандартного базиса преобразуются в векторы с положительными координатами (2, 1, 1), (1, 2, 1) и (1, 1, 2) соответственно.



Как линейное преобразование, определяемое этой матрицей, влияет на векторы стандартного базиса

Линейная комбинация новых векторов с положительными скалярами вытягивает изображение в направлениях $+x$, $+y$ и $+z$ больше, чем такая же линейная комбинация стандартного базиса.

Упражнение 5.7. Реализуйте функцию `multiple_matrix_vector` иначе — с использованием двух вложенных генераторов списков: один должен выполнять обход строк матрицы, другой — обход элементов строк.

Решение

```
def multiply_matrix_vector(matrix,vector):
    return tuple(
        sum(vector_entry * matrix_entry
            for vector_entry, matrix_entry in zip(row,vector))
        for row in matrix
    )
```

Упражнение 5.8. Реализуйте `multiple_matrix_vector` еще одним способом, используя то, что выходные координаты являются скалярными произведениями строк входной матрицы на входной вектор.

Решение. Это упрощенная версия решения предыдущего упражнения:

```
def multiply_matrix_vector(matrix,vector):
    return tuple(
        dot(row,vector)
        for row in matrix
    )
```

Упражнение 5.9. Мини-проект. Я рассказал вам, что такое линейное преобразование, и показал, что любое линейное преобразование можно описать матрицей. Теперь попробуйте доказать обратное — что все матрицы описывают линейные преобразования. Приведите алгебраическое доказательство, начав с явных формул умножения двухмерного вектора на матрицу 2×2 или трехмерного вектора на матрицу 3×3. То есть покажите, что матричное умножение сохраняет векторную сумму и произведение вектора на скаляр.

Решение. Я покажу доказательство для двух измерений, но доказательство для трех измерений имеет ту же структуру, только немного длиннее. Предположим, у нас есть матрица 2×2 с именем A, состоящая из четырех

произвольных чисел a, b, c и d. Посмотрим, как A воздействует на векторы $\mathbf{u}$ и $\mathbf{v}$:

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}; \ \mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}; \ \mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}.$$

Мы можем выполнить умножение матриц, чтобы найти $A\mathbf{u}$ и $A\mathbf{v}$:

$$A\mathbf{u} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} au_1 + bu_2 \\ cu_1 + du_2 \end{pmatrix};$$

$$A\mathbf{v} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} av_1 + bv_2 \\ cv_1 + dv_2 \end{pmatrix},$$

затем вычислить $A\mathbf{u} + A\mathbf{v}$ и $A(\mathbf{u} + \mathbf{v})$ и сравнить результаты:

$$A\mathbf{u} + A\mathbf{v} = \begin{pmatrix} au_1 + bu_2 \\ cu_1 + du_2 \end{pmatrix} + \begin{pmatrix} av_1 + bv_2 \\ cv_1 + dv_2 \end{pmatrix} = \begin{pmatrix} au_1 + av_1 + bu_2 + bv_2 \\ cu_1 + cv_1 + du_2 + dv_2 \end{pmatrix};$$

$$A(\mathbf{u} + \mathbf{v}) = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} u_1 + v_1 \\ u_2 + v_2 \end{pmatrix} = \begin{pmatrix} a(u_1 + v_1) + b(u_2 + v_2) \\ c(u_1 + v_1) + d(u_2 + v_2) \end{pmatrix} =$$

$$= \begin{pmatrix} au_1 + av_1 + bu_2 + bv_2 \\ cu_1 + cv_1 + du_2 + dv_2 \end{pmatrix}.$$

То есть двухмерное векторное преобразование, определяемое *любой* матрицей 2×2, сохраняет векторную сумму. Аналогично, для произведения вектора на произвольный скаляр s имеем:

$$s\mathbf{v} = \begin{pmatrix} sv_1 \\ sv_2 \end{pmatrix};$$

$$s \cdot (A\mathbf{v}) = \begin{pmatrix} s(av_1 + bv_2) \\ s(cv_1 + dv_2) \end{pmatrix} = \begin{pmatrix} sav_1 + sbv_2 \\ scv_1 + sdv_2 \end{pmatrix};$$

$$A(s\mathbf{v}) = \begin{pmatrix} a(sv_1) + b(sv_2) \\ c(sv_1) + d(sv_2) \end{pmatrix} = \begin{pmatrix} sav_1 + sbv_2 \\ scv_1 + sdv_2 \end{pmatrix}.$$

То есть $s \cdot (A\mathbf{v})$ и $A(s\mathbf{v})$ дают одинаковые результаты, соответственно, умножение на любую матрицу A сохраняет также результат произведения на скаляр. Эти два факта означают, что умножение на любую матрицу 2×2 является линейным преобразованием двухмерных векторов.

Упражнение 5.10. Снова воспользуемся двумя матрицами из подраздела 5.1.3:

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & -1 & 1 \end{pmatrix}; \ B = \begin{pmatrix} 0 & 2 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix}.$$

Напишите функцию `compose_a_b`, которая создает композицию линейных преобразований A и B. Затем примените функцию `infer_matrix` из упражнения 5.1, чтобы показать, что `infer_matrix(3, compose_a_b)` совпадает с произведением матриц AB.

Решение. Сначала реализуем две функции, `transform_a` и `transform_b`, которые выполняют линейные преобразования, определенные матрицами A и B. Затем объединим их с помощью функции `compose`:

```
from transforms import compose

a = ((1,1,0),(1,0,1),(1,-1,1))
b = ((0,2,1),(0,1,0),(1,0,-1))

def transform_a(v):
    return multiply_matrix_vector(a,v)

def transform_b(v):
    return multiply_matrix_vector(b,v)

compose_a_b = compose(transform_a, transform_b)
```

Затем используем функцию `infer_matrix`, чтобы найти матрицу, соответствующую этой композиции линейных преобразований, и сравнить ее с матричным произведением AB:

```
>>> infer_matrix(3, compose_a_b)
((0, 3, 1), (1, 2, 0), (1, 1, 0))
>>> matrix_multiply(a,b)
((0, 3, 1), (1, 2, 0), (1, 1, 0))
```

Упражнение 5.11. Мини-проект. Найдите две матрицы 2×2, не являющиеся единичными матрицами I_2, но произведение которых — единичная матрица.

Решение. Один из способов — подобрать элементы матриц методом простого перебора. Другой — рассмотреть задачу с точки зрения линейных преобразований. Если произведение двух матриц дает единичную матрицу, то композиция соответствующих им линейных преобразований должна давать тождественное преобразование.

Композиция каких двух двухмерных линейных преобразований является тождественным преобразованием? При последовательном применении к заданному двухмерному вектору эти линейные преобразования должны дать в результате исходный вектор. Одной из таких пар преобразований являются повороты на 90° и 270° по часовой стрелке. Последовательное применение обоих этих преобразований повернет исходный вектор на 360° и установит его в исходное положение. Матрицы для поворота на 270° и 90° показаны далее, а их произведение — это единичная матрица:

$$\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Упражнение 5.12. Квадратную матрицу можно умножить на саму себя любое количество раз. Такое последовательное умножение матрицы на саму себя можно рассматривать как возведение в степень. Для квадратной матрицы A произведение AA можно записать как A^2, AAA — как A^3 и т. д. Напишите функцию `matrix_power(power,matrix)`, которая возводит матрицу `matrix` в степень `power` (целое число).

Решение. Вот реализация, которая выполняет возведение матрицы в целую степень 1 и выше:

```
def matrix_power(power,matrix):
    result = matrix
    for _ in range(1,power):
        result = matrix_multiply(result,matrix)
    return result
```

5.2. ИНТЕРПРЕТАЦИЯ МАТРИЦ РАЗНОЙ ФОРМЫ

Функция `matrix_multiply` не зависит от размера входных матриц, поэтому ее можно использовать для умножения матриц 2×2 или 3×3. Но, как оказывается, она способна работать с матрицами и других размеров. Например, может перемножить две матрицы 5×5:

```
>>> a = ((-1, 0, -1, -2, -2), (0, 0, 2, -2, 1), (-2, -1, -2, 0, 1),
(0, 2, -2,-1, 0), (1, 1, -1, -1, 0))
>>> b = ((-1, 0, -1, -2, -2), (0, 0, 2, -2, 1), (-2, -1, -2, 0, 1),
(0, 2, -2,-1, 0), (1, 1, -1, -1, 0))
>>> matrix_multiply(a,b)
((-10, -1, 2, -7, 4), (-2, 5, 5, 4, -6), (-1, 1, -4, 2, -2),
(-4, -5, -5, -9, 4), (-1, -2, -2, -6, 4))
```

Давайте воспринимать этот результат всерьез — функции сложения векторов, умножения вектора на скаляр, скалярного произведения векторов и умножения матриц не зависят от размерности векторов. Конечно, мы не можем изобразить пятимерный вектор, но можем выполнить все те же алгебраические действия с пятерками чисел, которые выполняли с парами и тройками в двух- и трехмерном пространстве соответственно. Элементы произведения пятимерных матриц по-прежнему являются скалярными произведениями строк первой матрицы на столбцы второй (рис. 5.5):

Пересечение четвертого столбца и второй строки

Четвертый столбец

Вторая строка

$$\begin{pmatrix} -1 & 0 & -1 & -1 & -1 \\ 0 & 0 & 2 & -2 & 1 \\ -2 & -1 & -2 & 0 & 1 \\ 0 & 2 & -2 & -1 & 0 \\ 1 & 1 & -1 & -1 & 0 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 & -1 & 2 \\ -1 & -2 & -1 & -2 & 0 \\ 0 & 1 & 2 & 2 & -2 \\ 2 & -1 & -1 & 1 & 0 \\ 2 & 1 & -1 & 2 & -2 \end{pmatrix} = \begin{pmatrix} -10 & -1 & 2 & -7 & 4 \\ -2 & 5 & 5 & 4 & -6 \\ -1 & 1 & -4 & 2 & -2 \\ -4 & -5 & -5 & -9 & 4 \\ -1 & -2 & -2 & -6 & 4 \end{pmatrix}$$

$$(0, 0, 2, -2, 1) \cdot (-1, -2, 2, 1, 2) = 4$$

Рис. 5.5. Скалярное произведение строки первой матрицы на столбец второй дает один элемент матричного произведения

Мы не можем визуализировать результат этого произведения как прежде, но можем алгебраически показать, что матрица 5×5 задает линейное преобразование пятимерных векторов. В следующей главе поговорим об объектах, существующих в пространствах с четырьмя, пятью и большим количеством измерений.

5.2.1. Векторы-столбцы как матрицы

Вернемся к примеру умножения матрицы на вектор-столбец. Я уже показывал, как выполнить такое умножение, но мы рассматривали его как отдельный случай на примере функции `multiply_matrix_vector`. Оказывается, `matrix_multiply` тоже может вычислять эти произведения, правда, при этом вектор-столбец должен передаваться в виде матрицы. Для демонстрации передадим функции `matrix_multiply` следующую квадратную матрицу и матрицу с одним столбцом:

$$C = \begin{pmatrix} -1 & -1 & 0 \\ -2 & 1 & 2 \\ 1 & 0 & -1 \end{pmatrix}; \; D = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

Раньше я утверждал, что вектор можно рассматривать как матрицу с одним столбцом, поэтому мы могли представить `d` как вектор `(1,1,1)`. Но на этот раз будем считать его матрицей, имеющей три строки, в каждой из которых по одному элементу. Обратите внимание на то, что строки в данном случае должны записываться как `(1,)`, а не `(1)`, чтобы Python интерпретировал их как одноэлементные кортежи, а не как числа в скобках:

```
>>> c = ((-1, -1, 0), (-2, 1, 2), (1, 0, -1))
>>> d = ((1,),(1,),(1,))
>>> matrix_multiply(c,d)
((-2,), (1,), (0,))
```

Результат состоит из трех строк, каждая из которых содержит один элемент, а значит, это тоже матрица с одним столбцом. Вот как это произведение выглядит в матричных обозначениях:

$$\begin{pmatrix} -1 & -1 & 0 \\ -2 & 1 & 2 \\ 1 & 0 & -1 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} -2 \\ 1 \\ 0 \end{pmatrix}.$$

Наша функция `multiply_matrix_vector` может вычислять те же произведения, но с использованием другого формата:

```
>>> multiply_matrix_vector(c,(1,1,1))
(-2, 1, 0)
```

Данный пример показывает, что умножение матрицы на вектор-столбец — это частный случай матричного умножения. Таким образом, нам не нужна отдельная функция `multiple_matrix_vector`. Можно также отметить, что элементы результата являются скалярными произведениями строк первой матрицы на единственный столбец второй матрицы (рис. 5.6).

$$\begin{pmatrix} -1 & -1 & 0 \\ 2 & 1 & 2 \\ 1 & 0 & -1 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} -2 \\ 1 \\ 0 \end{pmatrix}$$

Рис. 5.6. Элементы результирующего вектора вычисляются как скалярное произведение

На бумаге не видно особой разницы между представлением в виде кортежей (с запятыми) и в виде вектора-столбца. Но для функций на Python, написанных нами, это различие имеет решающее значение. Кортеж (−2, 1, 0) нельзя использовать вместо кортежа кортежей ((−2,), (1,), (0,)). Другой способ записи того же вектора — запись в форме *вектора-строки*, или матрицы с одной строкой. В табл. 5.1 приводятся три обозначения для сравнения.

Таблица 5.1. Сравнение математических форм записи векторов и соответствующих представлений на Python

Представление	Форма записи в математике	На Python
Упорядоченная тройка (упорядоченный кортеж)	$\mathbf{v} = (-2, 1, 0)$	`v = (-2, 1, 0)`
Вектор-столбец	$\mathbf{v} = \begin{pmatrix} -2 \\ 1 \\ 0 \end{pmatrix}$	`v = ((-2,),(1,),(0,))`
Вектор-строка	$\mathbf{v} = (-2, 1, 0)$	`v = ((-2,1,0),)`

Увидев такое сравнение на уроке математики, можно было бы подумать, что это чрезмерно педантичное описание различий в обозначениях. Однако, если представить их на языке Python, становится очевидно, что в действительности это три разных объекта, обращаться с которыми нужно по-разному. Все они представляют одни и те же геометрические данные — стрелку или точку в трехмерном пространстве, но только на один из них, а именно на вектор-столбец, можно умножить матрицу 3×3. Вектор-строка не подходит, потому что, как показано на рис. 5.7, не получится найти скалярное произведение строки первой матрицы и столбца второй.



Рис. 5.7. Две матрицы, которые нельзя перемножить

Согласно определению умножения матриц матрицу можно умножить только на вектор-столбец, расположенный справа. Отсюда возникает общий вопрос, который мы рассмотрим в следующем разделе.

5.2.2. Какие пары матриц можно перемножить?

Мы можем составлять таблицы чисел любых размеров. В каких случаях наша формула умножения матриц работает и что это означает?

Формула умножения применима, когда количество столбцов первой матрицы совпадает с количеством строк второй. Это особенно очевидно, когда матричное умножение выполняется путем вычисления скалярных произведений. Например, можно умножить любую матрицу с тремя столбцами на матрицу с тремя строками. Это означает, что строки первой матрицы и столбцы второй содержат по три элемента, поэтому можно вычислить их скалярные произведения.

На рис. 5.8 показано скалярное произведение первой строки первой матрицы на первый столбец второй матрицы, дающее элемент в матрице произведения.

Первая строка

Первый столбец

$$\begin{pmatrix} 1 & -2 & 0 \\ -1 & -2 & 2 \end{pmatrix} \begin{pmatrix} 2 & 0 & -1 & 2 \\ 0 & -2 & 2 & -2 \\ -1 & -1 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 2 & ? & ? & ? \\ ? & ? & ? & ? \end{pmatrix}$$

Элемент на пересечении первой строки и первого столбца

Рис. 5.8. Вычисление первого элемента матрицы произведения

Можно продолжить вычисление этого матричного произведения и определить оставшиеся семь скалярных произведений. На рис. 5.9 показано вычисление еще одного элемента как скалярного произведения.

Вторая строка

Третий столбец

$$\begin{pmatrix} 1 & -2 & 0 \\ -1 & -2 & 2 \end{pmatrix} \begin{pmatrix} 2 & 0 & -1 & 2 \\ 0 & -2 & 2 & -2 \\ -1 & -1 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 4 & 3 & 6 \\ -4 & 2 & 1 & 4 \end{pmatrix}$$

Элемент на пересечении второй строки и третьего столбца

Рис. 5.9. Вычисление еще одного элемента матрицы произведения

Это ограничение также имеет смысл с точки зрения нашего первоначального определения матричного умножения: каждый столбец результата — это линейная комбинация столбцов первой матрицы со скалярами, заданными строками второй матрицы (рис. 5.10).

Три скаляра

$$\begin{pmatrix} 1 & -2 & 0 \\ -1 & -2 & 2 \end{pmatrix} \begin{pmatrix} 2 & 0 & -1 & 2 \\ 0 & -2 & 2 & -2 \\ -1 & -1 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 4 & 3 & 6 \\ -4 & 2 & 1 & 4 \end{pmatrix}$$

Три вектора-столбца

Столбец результата

$$\begin{pmatrix} 1 \\ -1 \end{pmatrix} \begin{pmatrix} -2 \\ -2 \end{pmatrix} \begin{pmatrix} 0 \\ 2 \end{pmatrix} \xrightarrow{\text{Линейная комбинация}} -1 \cdot \begin{pmatrix} 1 \\ -1 \end{pmatrix} + 2 \cdot \begin{pmatrix} -2 \\ -2 \end{pmatrix} + 2 \cdot \begin{pmatrix} 0 \\ 2 \end{pmatrix} = \begin{pmatrix} 3 \\ 1 \end{pmatrix}$$

Рис. 5.10. Каждый столбец результата — это линейная комбинация столбцов первой матрицы

Выше мы имели дело с квадратными матрицами 2×2 и 3×3, но в предыдущем примере (рис. 5.10) показано умножение матриц 2 на 3 и 3 на 4. Описывая размеры такой *прямоугольной* матрицы, сначала указывают количество строк, а затем количество столбцов. Например, трехмерный вектор-столбец — это матрица 3×1.

Используя такой способ обозначения, можно сформулировать общее утверждение о формах матриц, которые можно перемножать: умножать матрицу с размерами $n \times m$ на матрицу с размерами $p \times q$ можно, только если $m = p$. Если это условие выполняется, то матрица результата будет иметь размеры $n \times q$. Например, матрицу 17×9 нельзя умножить на матрицу 6×11. Однако матрицу 5×8 можно умножить на матрицу 8×10. На рис. 5.11 показан результат такого умножения — матрица 5×10.

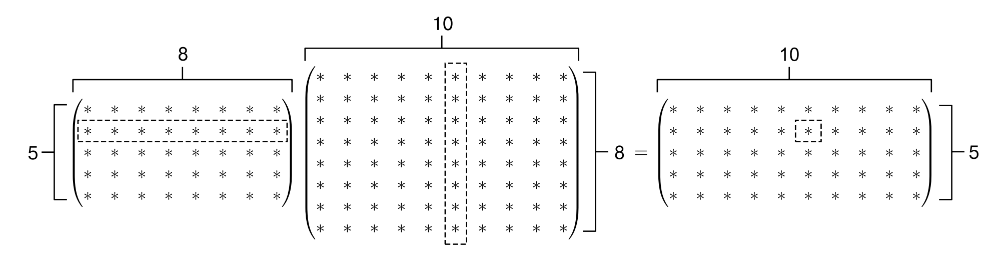


Рис. 5.11. Каждую из пяти строк первой матрицы можно умножить на любой из десяти столбцов второй матрицы, чтобы получить один из $5 \cdot 10 = 50$ элементов матрицы-произведения. Здесь вместо чисел использованы звездочки, чтобы показать, что любые матрицы этих размеров совместимы

Умножить эти матрицы в обратном порядке невозможно: матрицу 10×8 нельзя умножить на матрицу 5×8. Теперь мы знаем, как перемножать большие матрицы, но что означают результаты умножения? Как оказывается, из результата можно кое-что узнать: *все* матрицы являются векторными функциями и все допустимые матричные произведения можно интерпретировать как композицию этих функций. Исследуем этот вопрос подробнее.

5.2.3. Квадратные и прямоугольные матрицы как векторные функции

Матрицу 2×2 можно рассматривать как данные, необходимые для выполнения заданного линейного преобразования двухмерного вектора. Это преобразование, изображенное на рис. 5.12 в виде машины, принимает двухмерный вектор и выдает двухмерный вектор.

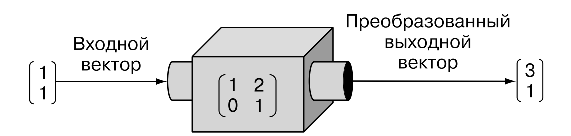


Рис. 5.12. Представление матрицы как машины, которая принимает и создает векторы

Внутри эта машина выполняет матричное умножение:

$$\begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 1 \end{pmatrix}.$$

О матрицах можно думать как о машинах, принимающих векторы на входе и производящих векторы на выходе. Однако, как показано на рис. 5.13, матрица не может принимать на входе любой вектор: поскольку матрица имеет размер 2×2, она выполняет линейное преобразование двухмерных векторов. Соответственно, ее можно умножить только на вектор-столбец с двумя элементами. Разделим вход и выход машины и предположим, что они принимают и производят двухмерные векторы, то есть пары чисел.

Точно так же машина линейного преобразования (рис. 5.14), основанная на матрице 3×3, может принимать и создавать только трехмерные векторы.

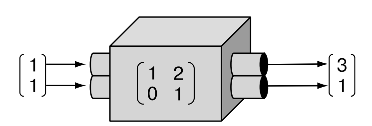


Рис. 5.13. Уточнение мысленной модели путем изменения представления о входах и выходах машины, показывающего, что ее входы и выходы — это пары чисел

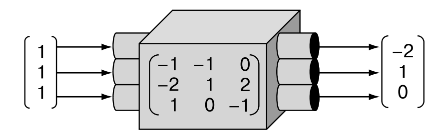


Рис. 5.14. Машина линейного преобразования, основанная на матрице 3×3, может принимать и создавать только трехмерные векторы

Теперь спросите себя: как выглядела бы машина, если бы она приводилась в действие прямоугольной матрицей? Например, такой:

$$\begin{pmatrix} -2 & -1 & -1 \\ 2 & -2 & 1 \end{pmatrix}.$$

То есть на какие векторы может умножаться матрица 2×3? Если говорить об умножении матрицы на вектор-столбец, то последний должен иметь три элемента, чтобы соответствовать размерам строк этой матрицы. Умножение матрицы 2×3 на вектор-столбец 3×1 дает в результате матрицу 2×1 — двухмерный вектор-столбец, например:

$$\begin{pmatrix} -2 & -1 & -1 \\ 2 & -2 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ -1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 3 \end{pmatrix}.$$

Таким образом, матрица 2×3 — это функция, преобразующая трехмерные векторы в двухмерные. Если изобразить ее как машину (рис. 5.15), то она выглядела бы как машина с тремя входами и двумя выходами.

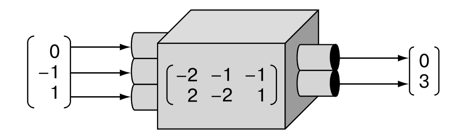


Рис. 5.15. Машина, основанная на матрице 2 × 3, принимает трехмерные и производит двухмерные векторы

В общем случае матрица размерами $m \times n$ определяет функцию, принимающую n-мерные и возвращающую m-мерные векторы. Любая такая функция линейна в том смысле, что она сохраняет векторные суммы и произведения векторов на скаляры. Это не преобразование, потому что функция не просто изменяет входной вектор, но возвращает вектор совершенно другого вида, имеющий другое количество измерений. По этой причине мы будем использовать более общую терминологию и называть матрицы *линейной функцией* или *линейным отображением*. Рассмотрим пример знакомого линейного отображения трехмерного объекта на двухмерную плоскость.

5.2.4. Проекция как линейное отображение трехмерного объекта на двухмерную плоскость

Мы уже видели функцию, которая принимает трехмерные и создает двухмерные векторы, проецируя трехмерные векторы на плоскость xy (см. подраздел 3.5.2). Это преобразование, назовем его P, принимает векторы в форме (x, y, z) и возвращает соответствующие им векторы без координаты z — (x, y). Далее я подробно расскажу, почему эта функция является линейным отображением и как она сохраняет векторную сумму и произведение вектора на скаляр.

Для начала запишем P как матрицу. Матрица преобразования, принимающего трехмерные и возвращающего двухмерные векторы, должна иметь размер 2×3. Воспользуемся проверенной формулой нахождения матрицы и исследуем действие P на векторы стандартного базиса. Напомню, что векторы стандартного базиса в трехмерном пространстве определяются как $\mathbf{e}_1 = (1, 0, 0)$, $\mathbf{e}_2 = (0, 1, 0)$ и $\mathbf{e}_3 = (0, 0, 1)$, когда к ним применяется проекция, в результате получаются векторы $(1, 0)$, $(0, 1)$ и $(0, 0)$ соответственно. Их можно записать в виде векторов-столбцов:

$$P(\mathbf{e}_1) = \begin{pmatrix} 1 \\ 0 \end{pmatrix};\ P(\mathbf{e}_2) = \begin{pmatrix} 0 \\ 1 \end{pmatrix};\ P(\mathbf{e}_3) = \begin{pmatrix} 0 \\ 0 \end{pmatrix},$$

а затем объединить, чтобы получить матрицу:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

Для проверки умножим ее тестовый вектор (a, b, c). Скалярное произведение (a, b, c) на $(1, 0, 0)$ равно a, и это первый элемент результата. Второй элемент — скалярное произведение (a, b, c) на $(0, 1, 0)$, то есть b. Вы можете представить эту матрицу как захват a и b из (a, b, c) и игнорирование c (рис. 5.16).

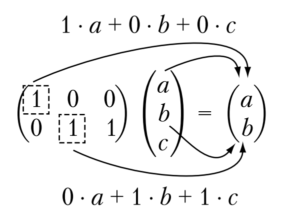


Рис. 5.16. Только $1 \cdot a$ вносит вклад в значение первого элемента произведения, и только $1 \cdot b$ вносит вклад в значение второго. Остальные элементы обнуляются

Эта матрица описывает нужное нам преобразование — удаляет третью координату из трехмерного вектора, оставляя только первые две координаты. Здорово, что мы можем записать операцию проекции в виде матрицы, но давайте рассмотрим еще и алгебраическое *доказательство* того, что это преобразование является линейным отображением. Для этого нужно показать, что выполняются два ключевых условия линейности.

Доказательство сохранения проекцией векторной суммы

Если P — это линейное преобразование, то оно должно обеспечивать сохранность любой векторной суммы $\mathbf{u} + \mathbf{v} = \mathbf{w}$. То есть $P(\mathbf{u}) + P(\mathbf{v})$ тоже должно равняться $P(\mathbf{w})$. Подтвердим это с помощью уравнений $\mathbf{u} = (u_1, u_2, u_3)$ и $\mathbf{v} = (v_1, v_2, v_3)$. Тогда $\mathbf{w} = \mathbf{u} + \mathbf{v}$, так что

$$\mathbf{w} = (u_1 + v_1, u_2 + v_2, u_3 + v_3).$$

Применение P ко всем этим векторам заключается в простом удалении третьей координаты:

$$P(\mathbf{u}) = (u_1, u_2);$$
$$P(\mathbf{v}) = (v_1, v_2),$$

то есть

$$P(\mathbf{w}) = (u_1 + v_1, u_2 + v_2).$$

Складывая $P(\mathbf{u})$ и $P(\mathbf{v})$, получаем сумму $(u_1 + v_1, u_2 + v_2)$, результат которой совпадает с $P(\mathbf{w})$. Следовательно, для любых трех трехмерных векторов $\mathbf{u} + \mathbf{v} = \mathbf{w}$ мы также имеем $P(\mathbf{u}) + P(\mathbf{v}) = P(\mathbf{w})$. Это подтверждает соблюдение первого условия.

Доказательство сохранения проекцией произведения вектора на скаляр

Теперь нужно показать, что P сохраняет произведение вектора на скаляр. Обозначив произвольное действительное число s и приняв $\mathbf{u} = (u_1, u_2, u_3)$, мы должны показать, что $P(s\mathbf{u})$ совпадает с $sP(\mathbf{u})$.

Удаление третьей координаты и умножение на скаляр дают один и тот же результат независимо от порядка выполнения этих операций. Результатом $s\mathbf{u}$ является $(s\mathbf{u}_1, s\mathbf{u}_2, s\mathbf{u}_3)$, поэтому $P(s\mathbf{u}) = (s\mathbf{u}_1, s\mathbf{u}_2)$. Результатом $P(\mathbf{u})$ является (u_1, u_2), поэтому $sP(\mathbf{u}) = (s\mathbf{u}_1, s\mathbf{u}_2)$. Это подтверждает соблюдение второго условия и то, что P удовлетворяет определению линейности.

Подобные доказательства обычно проще обосновывать, чем изучать, поэтому в упражнениях я предложу вам проверить еще одно доказательство, чтобы попрактиковаться: используя тот же подход, вы должны будете показать, что функция преобразования двухмерных векторов в трехмерные, заданная конкретной матрицей, линейна.

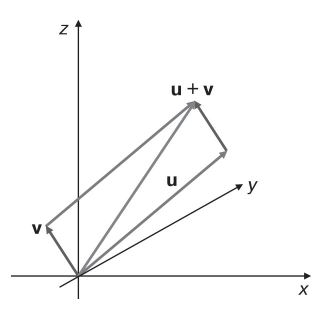


Рис. 5.17. Сумма произвольных векторов **u** и **v** в трехмерном пространстве

Но никакое доказательство не сравнится по наглядности с примером. Посмотрим, как выглядит результат проецирования трехмерной векторной суммы в двухмерную. Для этого выполним три шага. Во-первых, нарисуем векторную сумму двух векторов, **u** и **v**, в трехмерном пространстве (рис. 5.17).

Мы можем провести линии, соответствующие векторам, на плоскости *xy*, чтобы показать, как они будут располагаться после проецирования (рис. 5.18).

А теперь убедимся, что новые векторы *по-прежнему* образуют векторную сумму (рис. 5.19).

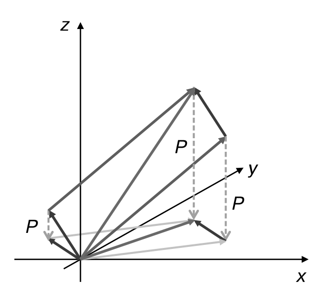


Рис. 5.18. Результат проецирования векторов **u**, **v** и **u** + **v** на плоскость *xy*

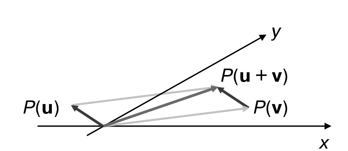


Рис. 5.19. Проекции векторов по-прежнему образуют векторную сумму: $P(\mathbf{v}) + P(\mathbf{v}) = P(\mathbf{u} + \mathbf{v})$

Иначе говоря, если три вектора, **u**, **v** и **w**, образуют векторную сумму $\mathbf{u} + \mathbf{v} = \mathbf{w}$, то их «тени» на плоскости *xy* также образуют векторную сумму. Теперь, когда вы получили некоторое представление о линейном преобразовании из трехмерного пространства в двухмерное и о матрице, которая его представляет, вернемся к обсуждению линейных отображений в целом.

5.2.5. Составление линейных отображений

Самое большое достоинство матриц в том, что они хранят все данные, необходимые для вычисления результата применения линейной функции к заданному вектору. Более того, размеры матрицы прямо сообщают о размерах входных и выходных векторов. Это визуально показано на рис. 5.20, где изображены машины на основе матриц разных размеров, имеющие разное количество входов и выходов. Эти четыре примера обозначены буквами, которые мы будем использовать в дальнейшем обсуждении.

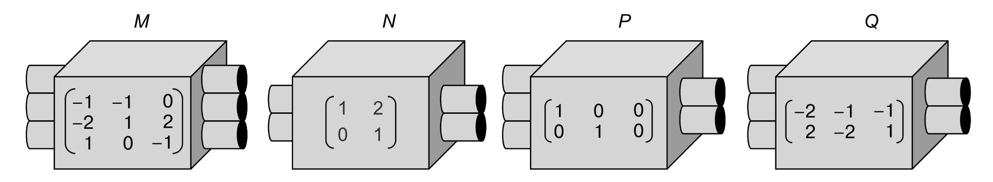


Рис. 5.20. Четыре линейные функции, изображенные в виде машин с разным количеством входов и выходов. Это количество сообщает нам размеры векторов, которые машины принимают или производят

Глядя на этот рисунок, легко определить, какие пары машин с линейными функциями можно соединить, чтобы построить новую машину. Например, количество выходов в M совпадает с количеством входов в P, поэтому мы можем составить композицию $P(M(\mathbf{v}))$ для трехмерного вектора $\mathbf{v}$. Машина M производит трехмерный вектор, который можно передать прямо в P (рис. 5.21).

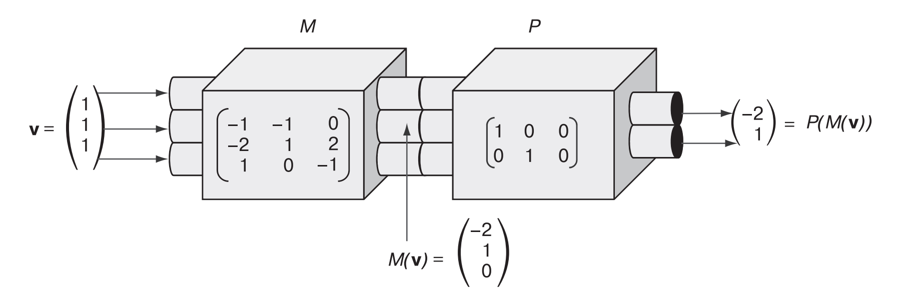


Рис. 5.21. Композиция P и M. Вектор подается на входы M, выходы $M(\mathbf{v})$ связаны незаметными соединениями с входами P, и на выходах P появляется результат композиции $P(M(\mathbf{v}))$

Для контраста на рис. 5.22 показана невозможная композиция N и M, потому что N не имеет достаточного количества выходов, чтобы загрузить все входы M.

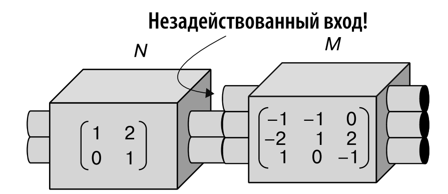


Рис. 5.22. Композиция *N* и *M* невозможна, потому что *N* выводит двухмерные векторы, тогда как машине *M* для работы нужны трехмерные векторы

Для наглядности я привел визуальный пример машин с разным числом входов и/или выходов, но, вообще говоря, он основан на более общем рассуждении, которое используется для определения возможности перемножить две матрицы. Количество столбцов матрицы слева должно совпадать с количеством строк в матрице справа. Когда эти количества совпадают, линейные функции можно объединить в композицию и перемножить их матрицы.

Представляя *P* и *M* как матрицы, мы можем записать композицию *P* и *M* как произведение матриц *PM*. (Напомню, что при применении композиции *PM* к вектору **v** как *PM***v** сначала применяется *M*, а затем *P*.) Когда **v** = (1, 1, 1), произведение *PM***v** превращается в произведение двух матриц и вектора-столбца, которое можно упростить до умножения одной матрицы *PM* на вектор-столбец (рис. 5.23).

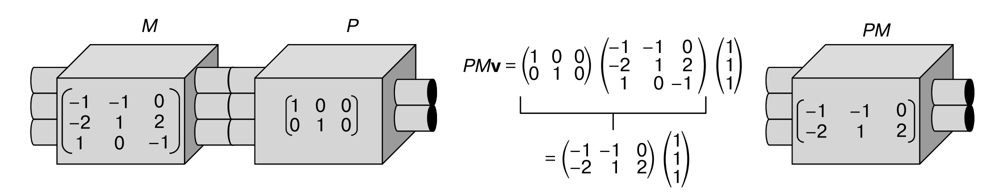


Рис. 5.23. Последовательное применение *M*, а затем *P* эквивалентно применению композиции *PM*. Композицию можно объединить в одну матрицу, выполнив матричное умножение

Программистам свойственно рассуждать о функциях с точки зрения типов данных, которые они принимают и возвращают. Ранее в этой главе я привел множество обозначений и терминов, и если вы разберетесь с основной концепцией, то в конце концов овладеете ею.

Я настоятельно рекомендую выполнить все упражнения из следующего раздела, чтобы окончательно понять язык матриц. В оставшейся части этой главы и в следующей будет не так много больших новых концепций — основное внимание мы сосредоточим на практическом применении того, что узнали к текущему моменту. Эти практические примеры помогут вам набить руку в матричных и векторных вычислениях.

5.2.6. Упражнения

Упражнение 5.13. Какие размеры имеет матрица

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{pmatrix}?$$

1. 5×3.
2. 3×5.

Решение. Это матрица 3×5: в ней три строки и пять столбцов.

Упражнение 5.14. Какие размеры имеет двухмерный вектор-столбец, если рассматривать его как матрицу? А двухмерный вектор-строка? Трехмерный вектор-столбец? Трехмерный вектор-строка?

Решение. Двухмерный вектор-столбец имеет две строки и один столбец, то есть это матрица 2×1. Двухмерный вектор-строка имеет одну строку и два столбца, то есть это матрица 1×2. Аналогично, трехмерные вектор-столбец и вектор-строка имеют размеры 3×1 и 1×3 соответственно, если рассматривать их как матрицы.

Упражнение 5.15. Мини-проект. Многие из реализованных нами векторных и матричных операций используют функцию `zip`. Когда она получает списки разной длины, то просто усекает более длинный, не генерируя ошибку. Это означает, что, передав ей недопустимые входные данные, мы получим бессмысленные результаты. Например, вычислить скалярное произведение двухмерного вектора и трехмерного в принципе невозможно, но функция `dot` все равно вернет некий результат:

```
>>> from vectors import dot
>>> dot((1,1),(1,1,1))
2
```

Добавьте защиту во все функции векторной арифметики, чтобы они генерировали исключения, получив векторы с несовместимыми размерами. После этого покажите, что `matrix_multiply` даже не пытается перемножить матрицы 3×2 и 4×5.

Упражнение 5.16. Какие из следующих матричных произведений действительные? Для действительных произведений укажите размерность матрицы результата.

1. $\begin{pmatrix} 10 & 0 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 8 & 2 & 3 & 6 \\ 7 & 8 & 9 & 4 \\ 5 & 7 & 0 & 9 \\ 3 & 3 & 0 & 2 \end{pmatrix}.$

2. $\begin{pmatrix} 0 & 2 & 1 & -2 \\ -2 & 1 & -2 & -1 \end{pmatrix} \begin{pmatrix} -3 & -5 \\ 1 & -4 \\ -4 & -4 \\ -2 & -4 \end{pmatrix}.$

3. $\begin{pmatrix} 1 \\ 3 \\ 0 \end{pmatrix} \begin{pmatrix} 3 & 3 & 5 & 1 & 3 & 0 & 5 & 1 \end{pmatrix}.$

4. $\begin{pmatrix} 9 & 2 & 3 \\ 0 & 6 & 8 \\ 7 & 7 & 9 \end{pmatrix} \begin{pmatrix} 7 & 8 & 9 \\ 10 & 7 & 8 \end{pmatrix}.$

Решение

Это недействительное произведение матриц 2×2 и 4×4: матрица слева имеет два столбца, а матрица справа — четыре строки.

Это *действительное* произведение матриц 2×4 и 4×2: четыре столбца матрицы слева соответствуют четырем строкам матрицы справа. В результате получается матрица 2×2.

Это *действительное* произведение матриц 3×1 и 1×8: единственный столбец матрицы слева соответствует единственной строке матрицы справа. В результате получается матрица 3×8.

Это недействительное произведение матриц 3×3 и 2×3: три столбца матрицы слева не соответствуют двум строкам матрицы справа.

Упражнение 5.17. Матрица, содержащая 15 элементов, умножается на матрицу с 6 элементами. Какие размеры должны иметь две исходные матрицы и матрица произведения?

Решение. Обозначим размерность матриц как $m \times n$ и $n \times k$, потому что количество столбцов в матрице слева должно совпадать с числом строк в матрице справа. Согласно условию задачи имеем $mn = 15$ и $nk = 6$. Этим равенствам соответствуют два набора значений:

- $m = 5$, $n = 3$ и $k = 2$, в этом случае имеет место произведение матрицы 5×3 на матрицу 3×2 и в результате получается матрица 5×2;
- $m = 15$, $n = 1$ и $k = 6$, в этом случае имеет место произведение матрицы 15×1 на матрицу 1×6 и в результате получается матрица 15×6.

Упражнение 5.18. Напишите функцию, которая превращает вектор-столбец в вектор-строку или наоборот. Такое переворачивание матрицы на бок называется *транспонированием*, а полученная матрица — *транспонированной.*

Решение

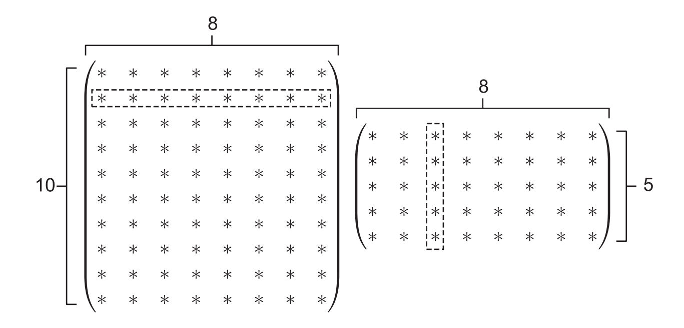
```
def transpose(matrix):
    return tuple(zip(*matrix))
```

Вызов `zip(*matrix)` возвращает список столбцов матрицы, который затем преобразуется в кортеж. Это приводит к замене векторов-строк векторами-столбцами в любой входной матрице, то есть векторы-столбцы превращаются в векторы-строки и наоборот:

```
>>> transpose(((1,),(2,),(3,)))
((1, 2, 3),)
>>> transpose(((1, 2, 3),))
((1,), (2,), (3,))
```

Упражнение 5.19. Нарисуйте схему, показывающую *невозможность* вычисления произведения матриц 10×8 и 5×8.

Решение



В матрице слева восемь столбцов, а в матрице справа пять строк. Это означает, что вычислить их произведение невозможно.

Упражнение 5.20. Нам нужно перемножить три матрицы: A размером 5×7, B размером 2×3 и C размером 3×5. В каком порядке их можно умножить и какого размера получится матрица-результат?

Решение. Одним из допустимых произведений является BC — произведение матриц 2×3 и 3×5, что дает матрицу 2×5. Другое допустимое произведение — CA — умножение матрицы 3×5 на матрицу 5×7, что дает матрицу 3×7. Произведение трех матриц BCA действительно независимо от порядка перемножения. $(BC)A$ — это произведение матриц 2×5 и 5×7, а $B(CA)$ — произведение матриц 2×3 и 3×7. В обоих случаях получается одна и та же матрица 2×7.

$$\underset{B}{\begin{pmatrix} 0 & -1 & 2 \\ -1 & -2 & -1 \end{pmatrix}} \underset{C}{\begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ -2 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 2 & -1 \end{pmatrix}} \underset{A}{\begin{pmatrix} 1 & -1 & 0 & -1 \\ -2 & -1 & 2 & -1 \\ -1 & 0 & -1 & -2 \\ 0 & -2 & 2 & -1 \\ 0 & -1 & -2 & -2 \end{pmatrix}}$$

Сначала B умножается на C

$$\underset{BC}{\begin{pmatrix} 2 & 3 & -2 & 4 & -2 \\ 4 & 0 & 1 & -2 & 0 \end{pmatrix}} \underset{A}{\begin{pmatrix} 1 & -1 & 0 & -1 \\ -2 & -1 & 2 & -1 \\ -1 & 0 & -1 & -2 \\ 0 & -2 & 2 & -1 \\ 0 & -1 & -2 & -2 \end{pmatrix}}$$

Сначала C умножается на A

$$\underset{B}{\begin{pmatrix} 0 & -1 & 2 \\ -1 & -2 & -1 \end{pmatrix}} \underset{CA}{\begin{pmatrix} -2 & -2 & 0 & -3 \\ 0 & 3 & -2 & 3 \\ -1 & -4 & 9 & 1 \end{pmatrix}}$$

$$\underset{BCA}{\begin{pmatrix} -2 & -11 & 20 & -1 \\ 3 & 0 & -5 & -4 \end{pmatrix}}$$

В обоих случаях получается один и тот же результат

Умножение трех матриц в разном порядке

Упражнение 5.21. Проекции на плоскости yz и xz тоже являются линейными отображениями из трехмерного пространства в двухмерное. Покажите, какие матрицы им соответствуют.

Решение. Проекция на плоскость yz удаляет координату x. Соответствующая матрица выглядит так:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Аналогично, проекция на плоскость xz удаляет координату y:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

например,

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \\ z \end{pmatrix}; \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} y \\ z \end{pmatrix}.$$

Упражнение 5.22. Покажите на примере, что функция `infer_matrix` из упражнения 5.1 может создавать матрицы для линейных функций, имеющих входы и выходы разных размерностей.

Решение. Одна из функций, которую можно использовать в роли примера, — проекция на плоскость xy, которая принимает трехмерные и возвращает двухмерные векторы. Это линейное преобразование можно реализовать как функцию на Python, а затем вывести его матрицу 2×3:

```
>>> def project_xy(v):
...     x,y,z = v
...     return (x,y)
...
>>> infer_matrix(3,project_xy)
((1, 0, 0), (0, 1, 0))
```

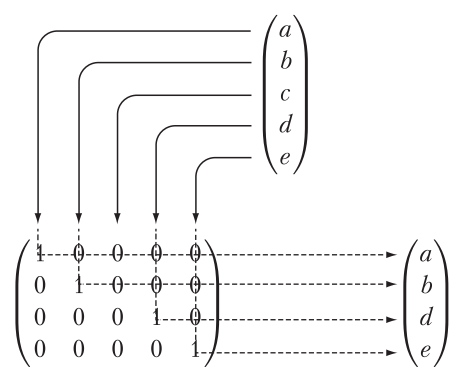
Обратите внимание: чтобы построить правильные векторы стандартного базиса для проверки `project_xy`, мы передали аргумент с размерностью *входных* векторов. Получив трехмерные векторы стандартного базиса, `project_xy` автоматически выводит двухмерные векторы, представляющие столбцы матрицы.

Упражнение 5.23. Покажите матрицу 4×5, которая воздействует на пятимерный вектор, удаляя третий элемент и создавая четырехмерный вектор. Например, умножение данной матрицы на вектор-столбец (1, 2, 3, 4, 5) должно дать в результате (1, 2, 4, 5).

Решение. Вот эта матрица:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Далее показано, как первая, вторая, четвертая и пятая координаты входного вектора образуют четыре координаты выходного вектора.



Единицы в матрице соответствуют координатам входного вектора, которые переносятся в выходной вектор

Упражнение 5.24. Мини-проект. Пусть есть вектор шести переменных (l, e, m, o, n, s). Найдите матрицу линейного преобразования, которая, воздействуя на этот вектор, даст в результате вектор (s, o, l, e, m, n).

Подсказка. Третья координата выходного вектора равна первой координате входного, поэтому вектор стандартного базиса (1, 0, 0, 0, 0, 0) должен быть преобразован в (0, 0, 1, 0, 0, 0).

Решение

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} l \\ e \\ m \\ o \\ n \\ s \end{pmatrix} = \begin{pmatrix} 0+0+0+0+0+s \\ 0+0+0+o+0+0 \\ l+0+0+0+0+0 \\ 0+e+0+0+0+0 \\ 0+0+m+0+n+0 \\ 0+0+0+0+1+0 \end{pmatrix} = \begin{pmatrix} s \\ o \\ l \\ e \\ m \\ n \end{pmatrix}.$$

Эта матрица переупорядочивает элементы шестимерного вектора, как определено в задании

Упражнение 5.25. Какие действительные произведения можно получить с матрицами M, N, P и Q из подраздела 5.2.5? Включите в ответы произведения матриц на самих себя. Укажите размеры матриц, получаемых в результате действительных произведений.

Решение. M имеет размер 3×3, N — 2×2, матрицы P и Q обе имеют размеры 2×3. Произведение M на саму себя, $MM = M^2$, допустимо и дает в результате матрицу 3×3, произведение $NN = N^2$ дает матрицу 2×2. Матрицы PM, QM, NP и NQ имеют размеры 3×2.

5.3. ПАРАЛЛЕЛЬНЫЙ ПЕРЕНОС ВЕКТОРОВ С ПОМОЩЬЮ МАТРИЦ

Одно из преимуществ матриц — сходство вычислений независимо от размеров. Нет необходимости беспокоиться о том, как выглядят векторы в двух- или трехмерном пространстве, — можно просто вставлять их в формулы умножения матриц или передавать в вызов функции `matrix_multiply`. Это особенно удобно, когда приходится выполнять вычисления с векторами, имеющими больше трех измерений.

Человеческий мозг не способен представить, как выглядят векторы в четырех или пяти измерениях, не говоря уже о 100, но мы уже видели, что без особого труда можем выполнять вычисления с векторами, имеющими большое число

измерений. В этом разделе рассмотрим вычисления, *требующие* расчетов в пространствах с большим числом измерений, — параллельный перенос векторов с использованием матрицы.

5.3.1. Придание линейности параллельному переносу

В предыдущей главе вы увидели, что параллельный перенос — это не линейное преобразование. При перемещении каждой точки плоскости на заданный вектор перемещается и начало координат, вследствие чего векторные суммы не сохраняются. Можно ли выполнить с помощью матрицы двухмерное преобразование, если оно нелинейное?

Хитрость заключается в том, чтобы представить параллельный перенос двухмерных точек как трехмерное преобразование. Вернемся к динозавру из главы 2. У нас имеется 21 точка, которые можно соединить отрезками и получить контур фигуры:

```
from vector_drawing import *

dino_vectors = [(6,4), (3,1), (1,2), (-1,5), (-2,5), (-3,4), (-4,4),
    (-5,3), (-5,2), (-2,2), (-5,1), (-4,0), (-2,1), (-1,0), (0,-3),
    (-1,-4), (1,-4), (2,-3), (1,-2), (3,-1), (5,1)
]

draw(
    Points(*dino_vectors),
    Polygon(*dino_vectors)
)
```

В результате получается уже знакомая двухмерная фигура динозавра (рис. 5.24).

Чтобы сдвинуть динозавра вправо на три единицы и вверх на одну единицу, можно просто прибавить вектор (3, 1) к каждой вершине. Но это нелинейное отображение, поэтому нельзя создать матрицу 2×2, выполняющую данный перенос. Если представить динозавра как обитателя трехмерного пространства, а не двухмерной плоскости, то окажется, что перенос можно выразить в виде матрицы.

Потерпите немного: я покажу одну хитрость, суть которой объясню чуть позже. Присвоим каждой точке динозавра координату z, равную 1. Теперь можно нарисовать фигуру в трехмерном пространстве, соединив точки отрезками. Как видите (рис. 5.25), получившийся многоугольник лежит на плоскости с $z = 1$. Я написал вспомогательную функцию `polygon_segments_3d`, чтобы нарисовать стороны многоугольника-динозавра в трехмерном пространстве:

```
from draw3d import *
def polygon_segments_3d(points,color='blue'):
    count = len(points)
    return [Segment3D(points[i], points[(i+1) % count],color=color)
    for i in range(0,count)]

dino_3d = [(x,y,1) for x,y in dino_vectors]

draw3d(
    Points3D(*dino_3d, color='blue'),
    *polygon_segments_3d(dino_3d)
)
```

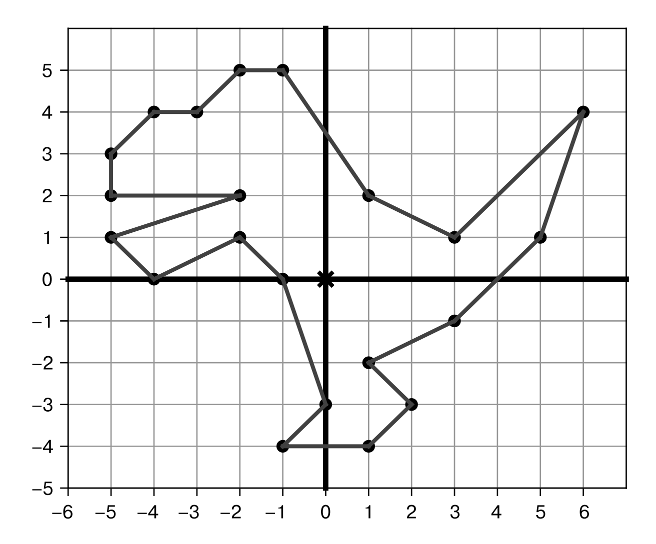


Рис. 5.24. Двухмерная фигура динозавра из главы 2

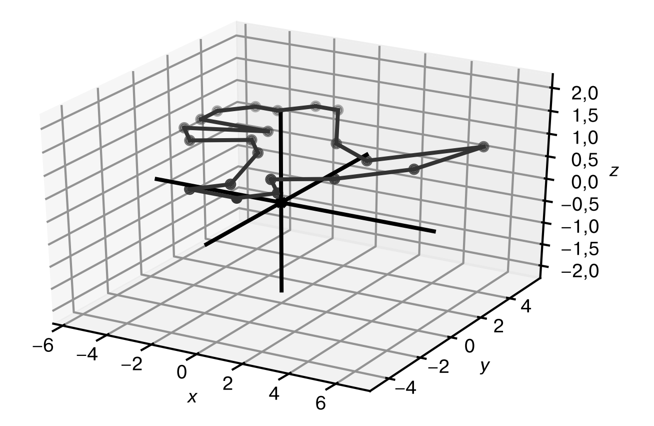


Рис. 5.25. Тот же динозавр, но каждая его вершина имеет координату $z = 1$

На рис. 5.26 показана матрица, которая искажает трехмерное пространство так, что начало координат остается на месте, а плоскость $z = 1$ перемещается в нужном направлении. Пока просто поверьте мне! Я выделил цифры, связанные с переносом, на которые следует обратить внимание.

$$\begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Рис. 5.26. Волшебная матрица, перемещающая плоскость $z = 1$ на +3 единицы вдоль оси x и на +1 единицу вдоль оси y

Мы можем применить эту матрицу к каждой вершине динозавра, и... вуаля! Динозавр сместится на (3, 1) в своей плоскости (рис. 5.27).

Вот код, применяющий матрицу:

```
magic_matrix = (
    (1,0,3),
    (0,1,1),
    (0,0,1))

translated = [multiply_matrix_vector(magic_matrix, v) for v in dino_vectors_3d]
```

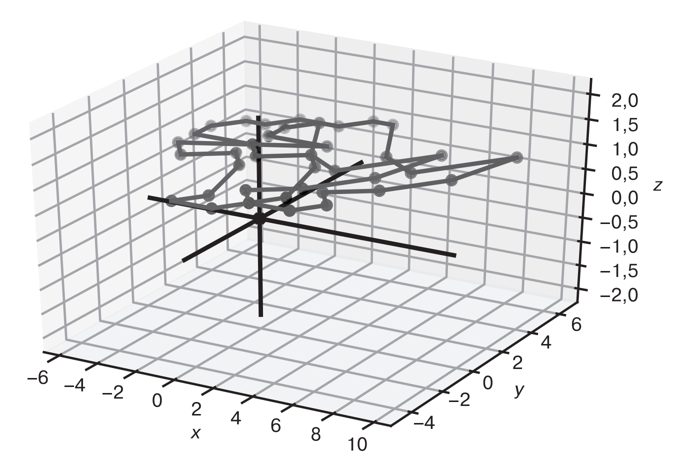


Рис. 5.27. Применение матрицы к каждой точке оставляет динозавра в плоскости, в которой он находится, но переносит на вектор (3, 1)

Для ясности можем удалить координату z и показать результат переноса динозавра в одной плоскости с исходным (рис. 5.28).

Можете опробовать приведенный мною код и убедиться, что конечное изображение динозавра действительно переместилось на вектор (3, 1). А теперь я расскажу, в чем фокус.

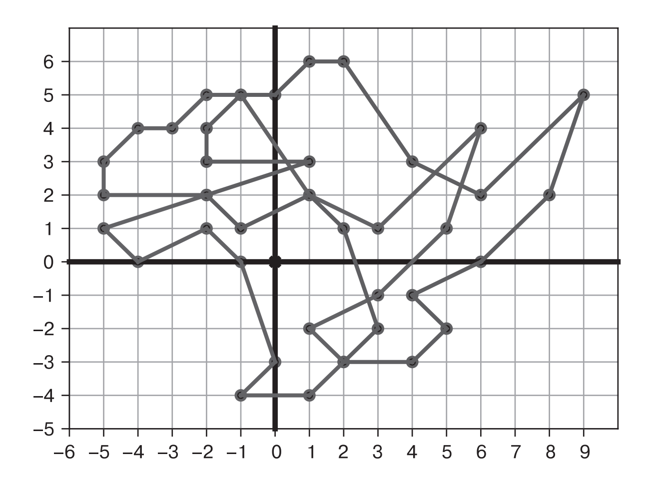


Рис. 5.28. После отображения результата переноса динозавра на двухмерную плоскость

5.3.2. Поиск трехмерной матрицы для двухмерного параллельного переноса

Столбцы нашей «волшебной» матрицы, как и столбцы любой матрицы, определяют, где окажутся векторы стандартного базиса после преобразования. Назовем эту матрицу T, тогда векторы $\mathbf{e}_1$, $\mathbf{e}_2$ и $\mathbf{e}_3$ будут преобразованы в векторы $T\mathbf{e}_1 = (1, 0, 0)$, $T\mathbf{e}_2 = (0, 1, 0)$ и $T\mathbf{e}_3 = (3, 1, 1)$. То есть векторы $\mathbf{e}_1$ и $\mathbf{e}_2$ остаются неизменными, а $\mathbf{e}_3$ изменяет только координаты x и y (рис. 5.29).

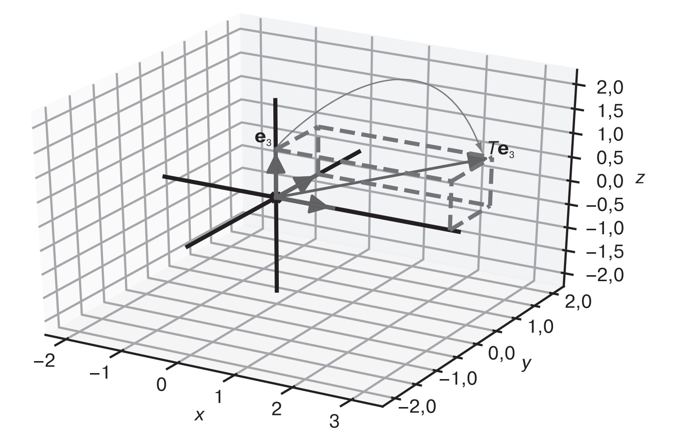


Рис. 5.29. Эта матрица перемещает только вектор $\mathbf{e}_3$, оставляя $\mathbf{e}_1$ и $\mathbf{e}_2$ неизменными

Любая точка в трехмерном пространстве и, следовательно, любая точка нашего динозавра определяется как линейная комбинация $\mathbf{e}_1$, $\mathbf{e}_2$ и $\mathbf{e}_3$. Например, кончик хвоста динозавра находится в точке (6, 4, 1), то есть $6\mathbf{e}_1 + 4\mathbf{e}_2 + \mathbf{e}_3$.

Поскольку T не влияет на $\mathbf{e}_1$ и $\mathbf{e}_2$, воздействие на $\mathbf{e}_3$ смещает точку $T(\mathbf{e}_3) = \mathbf{e}_3 + (3, 1, 0)$ так, что она переносится на +3 вдоль оси x и на +1 вдоль оси y. Это можно подтвердить алгебраически. Любой вектор $(x, y, 1)$ преобразуется в (3, 1, 0) с помощью матрицы

$$\begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} 1x + 0y + 3\cdot 1 \\ 0x + 1y + 1\cdot 1 \\ 0x + 0y + 1\cdot 1 \end{pmatrix} = \begin{pmatrix} x+3 \\ y+1 \\ 1 \end{pmatrix}.$$

В общем случае для переноса набора двухмерных векторов на некоторый вектор (a, b) нужно выполнить следующие шаги.

1. Переместить двухмерные векторы на плоскость $z = 1$ в трехмерном пространстве, чтобы каждый получил координату z, равную 1.
2. Умножить векторы на матрицу с выбранными значениями a и b:

$$\begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix}.$$

3. Удалить координату z из всех векторов, вернув их в двухмерное пространство.

Теперь, научившись выполнять параллельный перенос с помощью матриц, можно попробовать объединить его с другими линейными преобразованиями.

5.3.3. Комбинирование параллельного переноса с другими линейными преобразованиями

В предыдущей матрице первые два столбца в точности соответствуют векторам $\mathbf{e}_1$ и $\mathbf{e}_2$, а это означает, что за параллельный перенос отвечает только $\mathbf{e}_3$. Нам нежелательно, чтобы $T(\mathbf{e}_1)$ и $T(\mathbf{e}_2)$ имели компоненту z, отличную от нуля, потому что это выведет фигуру из плоскости $z = 1$. Но мы можем изменять другие компоненты (рис. 5.30).

Как оказывается, в верхний левый угол можно поместить любую матрицу 2×2 (как показано на рис. 5.30), описывающую соответствующее линейное преобразование, *дополняющее* параллельный перенос, определяемый третьим столбцом. Например, матрица

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

выполняет поворот на 90° против часовой стрелки. Вставив ее в матрицу переноса, как показано на рис. 5.31, мы получим новую матрицу, которая поворачивает плоскость xy на 90°, а затем сдвигает ее на вектор (3, 1).

Чтобы увидеть, как работает это преобразование, применим его ко всем вершинам трехмерного динозавра на Python. На рис. 5.32 показан результат работы следующего кода:

```
rotate_and_translate = ((0,-1,3),(1,0,1),(0,0,1))
rotated_translated_dino = [multiply_matrix_vector(rotate_and_translate, v)
                           for v in dino_vectors_3d]
```



Рис. 5.30. Посмотрим, что произойдет, если переместить $T(\mathbf{e}_1)$ и $T(\mathbf{e}_2)$ в плоскости xy

$$\begin{pmatrix} 0 & -1 & 3 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Рис. 5.31. Матрица, поворачивающая $\mathbf{e}_1$ и $\mathbf{e}_2$ на 90° и выполняющая параллельный перенос на вектор (3, 1). Любая фигура на плоскости $z = 1$ претерпит оба преобразования

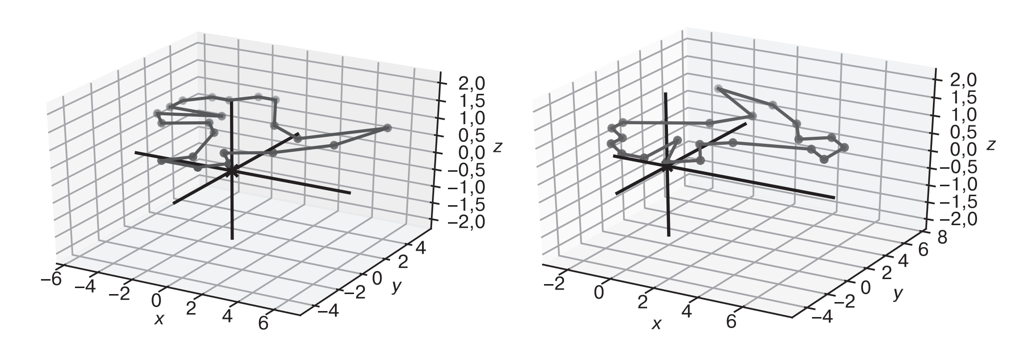


Рис. 5.32. Динозавр: *слева* — исходный; *справа* — преобразованный после поворота и параллельного переноса, вызванных применением единственной матрицы

Научившись выполнять двухмерный параллельный перенос с помощью матрицы, вы сможете задействовать этот подход для реализации трехмерного параллельного переноса. Для этого вам понадобится использовать матрицу 4×4 и войти в таинственный четырехмерный мир.

5.3.4. Параллельный перенос трехмерных объектов в четырехмерном мире

Что такое четвертое измерение? Четырехмерный вектор можно представить как стрелку с некоторой длиной, шириной, глубиной и еще одним измерением. Когда мы строили трехмерное пространство на основе двухмерного, то просто добавили координату z. Это означает, что трехмерные векторы могут находиться в плоскости xy, где $z = 0$, или в любой другой параллельной плоскости, где z имеет другое постоянное значение. На рис. 5.33 показаны некоторые из этих параллельных плоскостей.

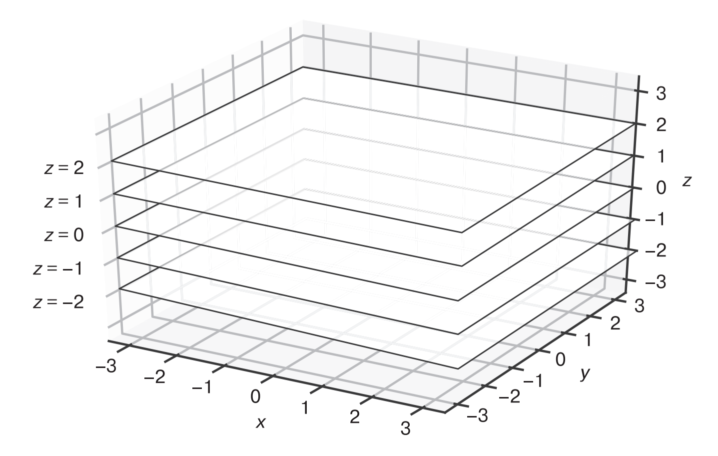


Рис. 5.33. Представление трехмерного пространства как стопки параллельных плоскостей, каждая из которых выглядит как плоскость *xy*, но имеет свою координату *z*

В соответствии с той же моделью четвертое измерение можно рассматривать как набор трехмерных пространств, которые индексируются некоторой четвертой координатой. Одна из возможных интерпретаций четвертой координаты — это время. Каждый снимок в данный момент времени представляет трехмерное пространство, а совокупность всех снимков — четвертое измерение, называемое *пространством-временем*. Начало пространства-времени — это начало пространства в момент времени $t = 0$ (рис. 5.34).

Это отправная точка теории относительности Эйнштейна. (Теперь вы достаточно подготовлены, чтобы взяться за чтение работ по этой теории, потому что она основана на четырехмерном пространстве-времени и линейных преобразованиях, заданных матрицами 4×4.)

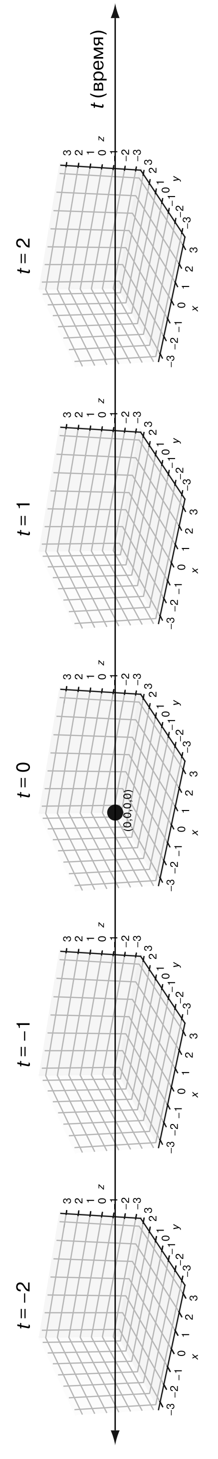


Рис. 5.34. Иллюстрация четырехмерного пространства-времени: аналогично тому, как срез трехмерного пространства с заданным значением z является двухмерной плоскостью, срез четырехмерного пространства-времени с заданным значением t — это трехмерное пространство

Векторная математика незаменима для высших размерностей, потому что с ростом количества измерений у нас быстро заканчиваются хорошие аналогии. Мне трудно представить пять, шесть, семь или более измерений, но математика координат для них ничуть не сложнее, чем для двух или трех измерений. Для наших текущих целей достаточно представлять четырехмерный вектор как набор из четырех чисел.

Повторим трюк, который сработал для параллельного переноса двухмерных векторов в трехмерном пространстве. Пусть есть трехмерный вектор (x, y, z), который нужно перенести на вектор (a, b, c). Мы можем прикрепить четвертую координату 1 к целевому вектору и использовать четырехмерную матрицу, описывающую параллельный перенос. Результат матричного умножения подтверждает, что мы получили желаемый результат (рис. 5.35).

Эта матрица увеличивает координату x на a, координату y — на b и координату z — на c, то есть выполняет преобразование, соответствующее переносу на вектор (a, b, c). Последовательность операций по добавлению четвертой координаты, применению матрицы 4×4 и последующему удалению четвертой координаты можно упаковать в функцию на Python:

$$\begin{pmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x+a \\ y+b \\ z+c \\ 1 \end{pmatrix}$$

Рис. 5.35. Добавив в вектор (x, y, z) четвертую координату, равную 1, мы можем выполнить параллельный перенос вектора на (a, b, c), используя эту матрицу

```
def translate_3d(translation):
    def new_function(target):
        a,b,c = translation
        x,y,z = target
        matrix = ((1,0,0,a),
                  (0,1,0,b),
                  (0,0,1,c),
                  (0,0,0,1))
        vector = (x,y,z,1)
        x_out, y_out, z_out, _ =\
            multiply_matrix_vector(matrix,vector)
        return (x_out,y_out,z_out)
    return new_function
```

Функция translate_3d принимает вектор, определяющий перенос, и возвращает новую функцию, которая применяет этот перенос к трехмерному вектору

Составление матрицы 4 × 4, описывающей перенос, а в следующей строке трехмерный вектор (x, y, z) превращается в четырехмерный добавлением четвертой координаты со значением 1

Применение четырехмерной матрицы преобразования

Наконец, нарисуем чайник и выполним параллельный перенос на вектор (2, 2, –3), чтобы проверить, сместится ли он так, как ожидается. Вы можете сделать это сами, запустив сценарий `matrix_translate_teapot.py`. У вас должно получиться такое же изображение, как на рис. 5.36.

Теперь, упаковав перенос в матричную операцию, мы можем объединить его с другими линейными трехмерными преобразованиями и применять за один шаг. Как оказывается, искусственную четвертую координату в этом подходе можно интерпретировать как время t.

Два изображения на рис. 5.36 можно считать снимками чайника в моменты времени $t = 0$ и $t = 1$, который движется в направлении $(2, 2, -3)$ с постоянной скоростью. Если вы ищете любопытную задачу, то можете заменить вектор $(x, y, z, 1)$ в этой реализации векторами в форме (x, y, z, t), где координата t меняется со временем. В моменты времени $t = 0$ и $t = 1$ положение чайника должно соответствовать кадрам, изображенным на рис. 5.36, а в промежутке чайник должен плавно перемещаться между этими местоположениями. Если вы сможете понять, как это работает, то приблизитесь к уровню Эйнштейна!

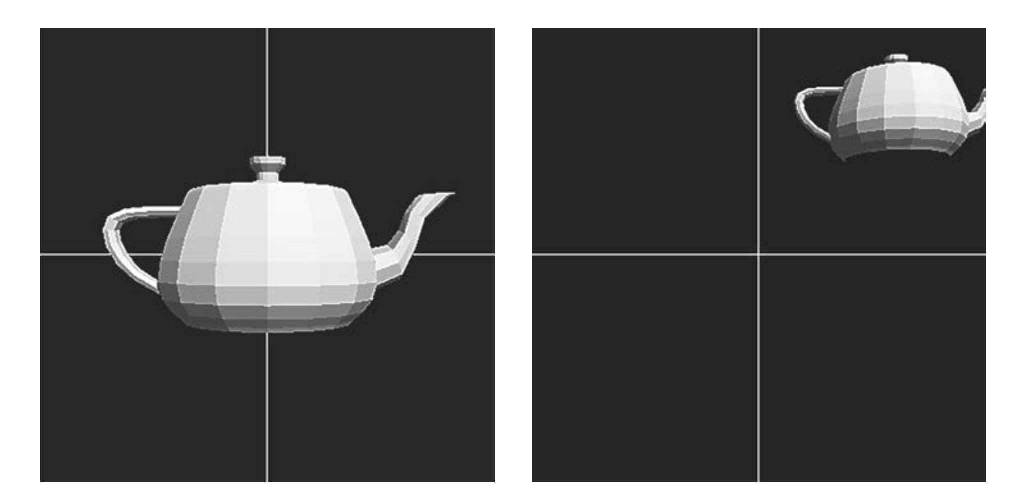


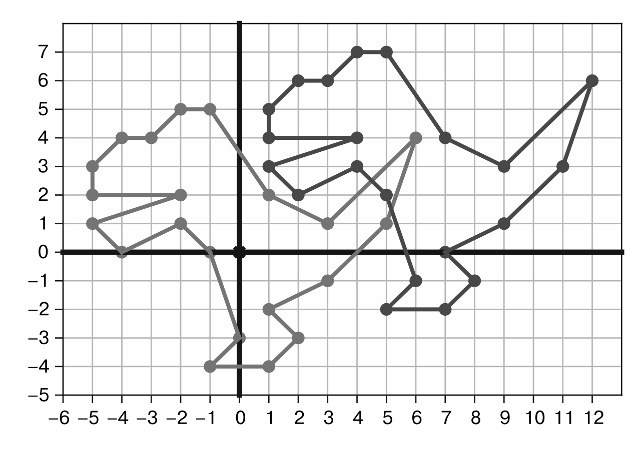
Рис. 5.36. Чайник: *слева* — до переноса; *справа* — после него. Как и ожидалось, в результате переноса чайник перемещается вверх, вправо и вдаль от наблюдателя

До сих пор мы рассматривали векторы исключительно как точки в пространстве, которые можно изобразить на экране компьютера. Это, конечно, важно, но такой подход не раскрывает всех возможностей, которые предоставляют векторы и матрицы. Изучение влияния линейных преобразований на векторы составляет суть *линейной алгебры*, и в следующей главе я дам более широкий обзор этого предмета вместе с некоторыми свежими примерами, имеющими отношение к программистам.

5.3.5. Упражнения

Упражнение 5.26. Покажите, что волшебство трехмерного матричного преобразования не работает, если поместить двухмерную фигуру, например динозавра, на плоскость $z = 2$. Что в действительности происходит?

Решение. Если использовать `[(x,y,2) for x,y in dino_vectors]` и применить ту же матрицу 3 × 3, то динозавр переместится в два раза дальше — на вектор (6, 2) вместо (3, 1). Это связано с тем, что вектор (0, 0, 1) переносится на (3, 1), а это линейное преобразование.



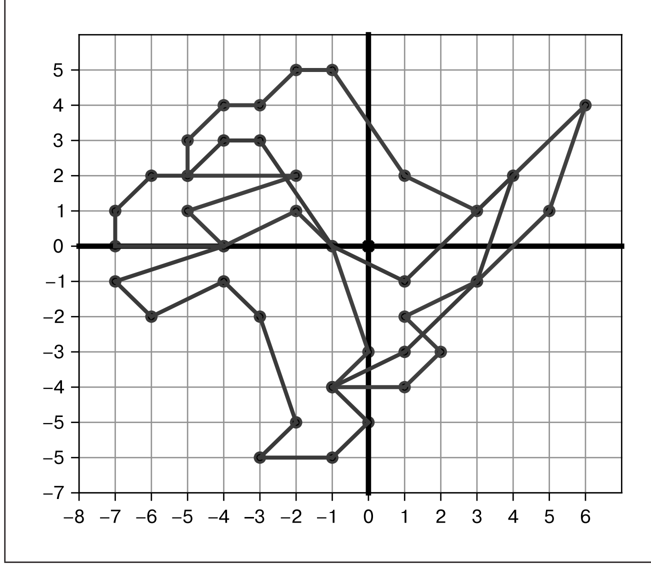
Динозавр в плоскости $z = 2$ переносится той же матрицей вдвое дальше

Упражнение 5.27. Придумайте матрицу для параллельного переноса динозавра на –2 единицы вдоль оси x и на –2 единицы вдоль оси y. Выполните преобразование и покажите результат.

Решение. Заменив значения 3 и 1 в исходной матрице на –2 и –2, получим:

$$\begin{pmatrix} 1 & 0 & -2 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix}.$$


Динозавр сместится вниз и влево на вектор (–2, –2).



Упражнение 5.28. Покажите, что любая матрица, подобная этой

$$\begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix},$$

не влияет на координату z трехмерного вектора-столбца, на который она умножается.

Решение. Если начальная координата z трехмерного вектора является числом z, то умножение на матрицу оставит эту координату неизменной:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} ax+by+cz \\ dx+ey+fz \\ 0x+0y+z \end{pmatrix}.$$

Упражнение 5.29. Мини-проект. Найдите матрицу 3 × 3, которая поворачивает двухмерную фигуру в плоскости $z = 1$ на 45°, уменьшает ее размеры в два раза и выполняет параллельный перенос на вектор (2, 2). Проверьте ее, применив к вершинам динозавра.

Решение. Сначала найдем матрицу 2 × 2 для поворота двухмерного вектора на 45°:

```
>>> from vectors import rotate2d
>>> from transforms import *
>>> from math import pi
>>> rotate_45_degrees = curry2(rotate2d)(pi/4)
>>> rotation_matrix = infer_matrix(2,rotate_45_degrees)
>>> rotation_matrix
((0.7071067811865476, -0.7071067811865475), (0.7071067811865475,
0.7071067811865476))
```

Вернет функцию, которая с помощью rotate2d выполняет поворот входного двухмерного вектора на угол 45° (или pi/4 рад)

Округлив до третьего десятичного знака, получаем матрицу

$$\begin{pmatrix} 0{,}707 & -0{,}707 \\ 0{,}707 & 0{,}707 \end{pmatrix}.$$

Аналогично находим матрицу, масштабирующую вектор с коэффициентом 1/2:

$$\begin{pmatrix} 0{,}5 & 0 \\ 0 & 0{,}5 \end{pmatrix}.$$

Перемножив эти матрицы, получаем матрицу, выполняющую сразу оба эти преобразования:

```
>>> from matrices import *
>>> scale_matrix = ((0.5,0),(0,0.5))
>>> rotate_and_scale = matrix_multiply(scale_matrix,rotation_matrix)
>>> rotate_and_scale
((0.3535533905932738, -0.35355339059327373), (0.35355339059327373,
0.3535533905932738))
```

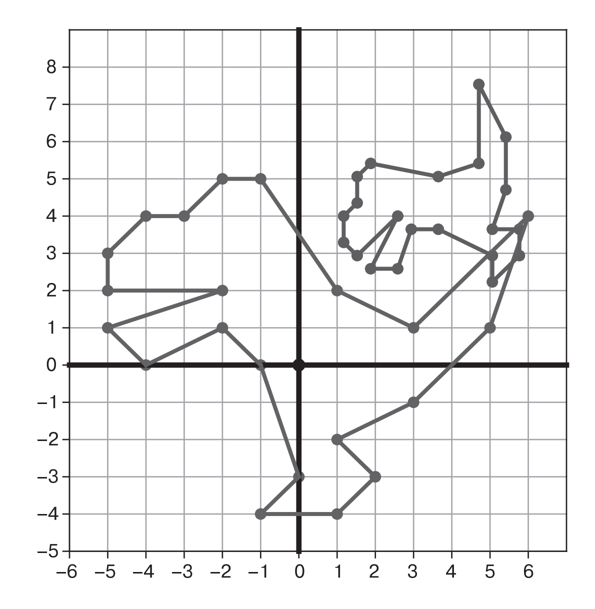
А вот матрица 3 × 3, которая выполняет параллельный перенос динозавра на вектор (2, 2) в плоскости $z = 1$:

$$\begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}.$$

В ее верхний левый угол можно вставить нашу матрицу 2×2 поворота и масштабирования и получить окончательную матрицу, которая нам нужна:

```
>>> ((a,b),(c,d)) = rotate_and_scale
>>> final_matrix = ((a,b,2),(c,d,2),(0,0,1))
>>> final_matrix
((0.3535533905932738, -0.35355339059327373, 2), (0.35355339059327373,
0.3535533905932738, 2), (0, 0, 1))
```

Переместив динозавра в плоскость $z = 1$, применив эту матрицу в трехмерном пространстве, а затем спроецировав результат обратно на двухмерную плоскость, получим уменьшенное изображение повернутого и смещенного динозавра. Все это было достигнуто единственным матричным умножением.



Упражнение 5.30. Матрица в предыдущем упражнении поворачивает динозавра на 45°, а затем переносит его на вектор (2, 2). Используя матричное умножение, постройте матрицу, которая выполняет сначала перенос на вектор (3, 1), а затем поворот на 90°.

Решение. Если предположить, что динозавр лежит на плоскости $z = 1$, то следующая матрица совершит поворот на 90° *без* переноса:

$$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Нам нужно сначала выполнить перенос динозавра, а затем повернуть его, поэтому умножим матрицу поворота на матрицу переноса:

$$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & -1 & -1 \\ 1 & 0 & 3 \\ 0 & 0 & 1 \end{pmatrix}.$$

Эта матрица отличается от той, которая выполняет сначала поворот, а потом перенос. В данном случае мы видим, что поворот на 90° влияет на вектор переноса (3, 1). Новый эффективный вектор переноса — (–1, 3).

Упражнение 5.31. Напишите функцию `translate_4d`, аналогичную функции `translate_3d`, которая использует матрицу 5×5 для параллельного переноса четырехмерного вектора на другой четырехмерный вектор. Запустите пример, чтобы показать, что координаты переводятся.

Решение. Суть здесь та же самая, только теперь размерность вектора увеличивается с 4 до 5 путем добавления пятой координаты 1:

```
def translate_4d(translation):
    def new_function(target):
        a,b,c,d = translation
        x,y,z,w = target
        matrix = (
            (1,0,0,0,a),
            (0,1,0,0,b),
            (0,0,1,0,c),
            (0,0,0,1,d),
            (0,0,0,0,1))
        vector = (x,y,z,w,1)
        x_out,y_out,z_out,w_out,_ = multiply_matrix_vector(matrix,vector)
        return (x_out,y_out,z_out,w_out)
    return new_function
```

Убедимся, что перенос выполняется (он дает тот же эффект, что и сложение векторов):

```
>>> translate_4d((1,2,3,4))((10,20,30,40))
(11, 22, 33, 44)
```

В предыдущих главах мы использовали визуальные примеры в двух- и трехмерном пространствах, чтобы наглядно показать, как работает векторная и матричная арифметика. По мере продвижения вперед мы стали уделять больше внимания вычислениям и в конце этой главы рассчитали векторные преобразования в более высоких размерностях, которые невозможно изобразить наглядно. Это одно из преимуществ линейной алгебры: она дает инструменты для решения геометрических задач, которые слишком сложны для визуального представления. В следующей главе рассмотрим широкий спектр примеров практического применения линейной алгебры.

КРАТКИЕ ИТОГИ ГЛАВЫ

- Линейное преобразование определяется его влиянием на векторы стандартного базиса. Векторы, получающиеся в результате применения линейного преобразования к стандартному базису, содержат всю информацию о преобразовании. Это означает, что для описания трехмерного линейного преобразования любого типа требуется всего девять чисел (по три координаты каждого из векторов результата). Для двухмерного линейного преобразования требуется четыре числа.
- В матричной записи линейное преобразование выражается в виде прямоугольной таблицы с числами. В соответствии с соглашениями матрица заполняется результатами применения преобразования к векторам стандартного базиса по столбцам.
- Применение матрицы для вычисления результата линейного преобразования, которое она представляет, называется *умножением матрицы на вектор*. При этом вектор обычно записывается как столбец его координат, сверху вниз, а не как кортеж.
- Две квадратные матрицы тоже можно перемножить. В результате образуется матрица, представляющая композицию линейных преобразований исходных матриц.
- Чтобы получить произведение двух матриц, нужно вычислить скалярные произведения строк первой матрицы на столбцы второй. Например, скалярное произведение строки i первой матрицы на столбец j второй матрицы дает значение в строке i и в столбце j матрицы произведения.
- Квадратные матрицы представляют линейные преобразования, а неквадратные матрицы — линейные функции от векторов одной размерности, дающие в результате векторы другой размерности. То есть эти функции преобразуют векторную сумму в векторную сумму, а произведение вектора на скаляр — в произведение вектора на скаляр.

- Размер матрицы сообщает размеры векторов, которые принимает и возвращает соответствующая линейная функция. Матрица с m строками и n столбцами называется матрицей $m \times n$. Она определяет линейную функцию, принимающую n-мерные векторы и возвращающую m-мерные векторы.
- Параллельный перенос — это *не* линейная функция, но его можно сделать линейным, если выполнять в более высоком измерении. Это позволяет выполнять переносы (одновременно с другими линейными преобразованиями) путем умножения матриц.

6 Обобщение до высших размерностей

В этой главе

- ✓ Реализация на Python обобщенного абстрактного базового класса для представления векторов.
- ✓ Определение векторных пространств и перечисление их полезных свойств.
- ✓ Интерпретация функций, матриц, изображений и звуковых волн как векторов.
- ✓ Поиск полезных подпространств в векторных пространствах, содержащих интересующие данные.

Даже если вас не интересует создание анимационных эффектов с чайниками, операции с векторами, линейными преобразованиями и матрицами все равно могут пригодиться. На самом деле эти концепции настолько полезны, что им посвящен целый подраздел математики — *линейная алгебра*. Она обобщает все, что мы знаем о двух- и трехмерной геометрии, и распространяет на данные с любым количеством измерений.

Программисты умеют обобщать идеи. При разработке сложного программного обеспечения часто приходится писать один и тот же код снова и снова. В какой-то момент вы осознаете это и объединяете код в один класс или функцию, способную обрабатывать все случаи, встречавшиеся до сих пор. Это избавляет

вас от необходимости делать лишнюю работу и часто улучшает организацию кода и удобство его сопровождения. Математики следуют тому же принципу: сталкиваясь с похожими закономерностями снова и снова, они могут точнее сформулировать свое видение и усовершенствовать определения.

В данной главе мы используем эту логику для определения *векторных пространств*. Векторные пространства — это наборы объектов, с которыми можно обращаться как с векторами. Это могут быть стрелки на плоскости, наборы чисел или объекты, совершенно отличные от тех, что мы видели до сих пор. Например, с изображениями тоже можно обращаться как с векторами и создавать их линейные комбинации (рис. 6.1).

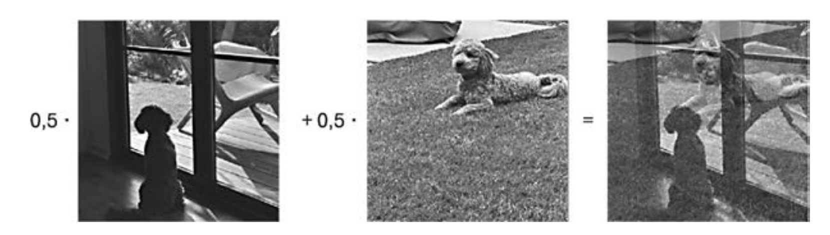


Рис. 6.1. Линейная комбинация двух изображений дает новое изображение

Ключевыми операциями в векторном пространстве являются сложение векторов и умножение вектора на скаляр. С их помощью можно составлять линейные комбинации, включая отрицание, вычитание, взвешивание и т. д., и рассуждать о том, какие преобразования являются линейными. Как оказывается, эти операции помогают понять значение слова *«размерность»*. Например, далее вы увидите, что изображения, приведенные на рис. 6.1, представляют собой 270 000-мерные объекты! Вскоре мы рассмотрим многомерные и даже бесконечномерные пространства, но не будем забегать вперед и начнем с обзора уже известных нам двух- и трехмерных пространств.

6.1. ОБОБЩЕНИЕ ОПРЕДЕЛЕНИЯ ВЕКТОРОВ

Язык Python поддерживает парадигму объектно-ориентированного программирования (ООП), что оказывается отличной основой для обобщения. В частности, классы поддерживают *наследование*, позволяющее создавать новые классы объектов, которые наследуют свойства и поведение существующего родительского класса. Далее мы попробуем реализовать уже знакомые двух- и трехмерные векторы как экземпляры более общего класса объектов, называемых просто векторами. После этого любые другие объекты, наследующие поведение родительского класса, тоже могут по праву называться векторами (рис. 6.2).

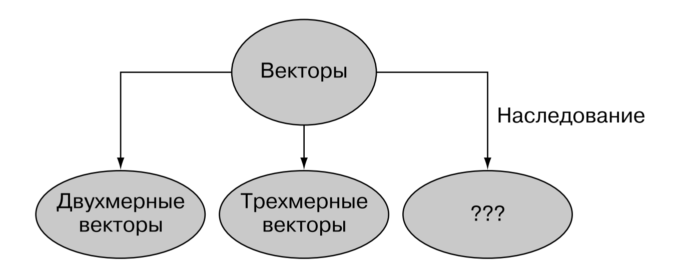


Рис. 6.2. Интерпретация двух- и трехмерных векторов и других объектов как частных случаев векторов

Если прежде вы не занимались объектно-ориентированным программированием или не использовали его в Python, не волнуйтесь. В этой главе я буду рассматривать самые простые случаи и помогу вам освоить их. Желающие узнать больше о классах и наследовании в языке Python, прежде чем приступить к работе, могут обратиться к приложению Б, где я представил довольно подробный обзор.

6.1.1. Создание класса векторов с двумя координатами

В примерах ранее двух- и трехмерные векторы были векторами *координат*, то есть определялись как кортежи чисел, представляющих координаты. (Мы также видели, что векторную арифметику можно определить геометрически в терминах стрелок, но этот подход нельзя воплотить непосредственно в код на Python.) В двухмерных векторах координат данные представлены упорядоченными парами координат x и y. Кортеж — отличный способ хранения таких данных, но мы можем использовать и класс. Назовем класс, представляющий двухмерные векторы координат, `Vec2`:

```
class Vec2():
    def __init__(self,x,y):
        self.x = x
        self.y = y
```

Мы можем инициализировать вектор инструкцией `v = Vec2 (1.6, 3.8)` и получить его координаты, обратившись к ним как к `v.x` и `v.y`. Можем добавить в этот класс методы, реализующие двухмерную векторную арифметику, такие как сложение и умножение на скаляр. Функция сложения `add` принимает второй вектор в аргументе и возвращает новый объект `Vec2`, чьи координаты представляют собой сумму координат x и y:

```
class Vec2():
    ...
    def add(self, v2):
        return Vec2(self.x + v2.x, self.y + v2.y)
```

При добавлении чего-то нового в существующий класс я иногда использую многоточие, замещающее существующий код

Вот как выглядит сложение векторов с помощью класса `Vec2`:



Здесь, как и в исходной реализации, сложение векторов выполняется не «на месте». То есть входные векторы не изменяются, а создается новый объект `Vec2` для хранения суммы. Аналогично можно реализовать метод умножения на скаляр, принимающий скаляр в аргументе и возвращающий новый вектор:

```
class Vec2():
    ...
    def scale(self, scalar):
        return Vec2(scalar * self.x, scalar * self.y)
```

Вызов `Vec2(1,1).scale(50)` вернет новый вектор с координатами x и y, равными 50. Есть еще кое-что, о чем нам нужно позаботиться: в настоящее время операция сравнения, такая как `Vec2(3,4) == Vec2(3,4)`, дает `False`. Это неправильно, потому что эти два экземпляра представляют один и тот же вектор. По умолчанию Python сравнивает экземпляры по ссылкам на них (проверяя, находятся ли они в одном и том же месте в памяти), а не по значениям. Это можно исправить, переопределив метод определения равенства, который вызывается, когда Python обрабатывает оператор `==` с объектами класса `Vec2` (подробнее этот аспект программирования на Python объясняется в приложении Б):

```
class Vec2():
    ...
    def __eq__(self,other):
        return self.x == other.x and self.y == other.y
```

Нам нужно, чтобы два двухмерных вектора координат считались равными, если совпадают их координаты x и y, и данное определение равенства обеспечивает это. После добавления метода операция `Vec2(3,4) == Vec2(3,4)` будет давать `True`.

Класс `Vec2` теперь поддерживает основные векторные операции — сложение и умножение на скаляр, а также правильно реагирует на операцию проверки равенства. Теперь можно обратить внимание на синтаксический сахар.

6.1.2. Усовершенствование класса Vec2

По аналогии с оператором `==` можно также изменить поведение операторов `+` и `*`, чтобы они выполняли сложение векторов и умножение вектора на скаляр соответственно. Этот прием называется *перегрузкой операторов* и описан в приложении Б:

```
class Vec2():
    ...
    def __add__(self, v2):
        return self.add(v2)
    def __mul__(self, scalar):
        return self.scale(scalar)
    def __rmul__(self, scalar):
        return self.scale(scalar)
```

Методы __mul__ и __rmul__ определяют два разных порядка следования сомножителей в операции умножения, благодаря чему можем реализовать поддержку умножения на скаляр слева и справа. С математической точки зрения оба порядка означают одно и то же

Теперь мы можем кратко записывать линейные комбинации. Например, `3.0 * Vec2(1,0) + 4.0 * Vec2(0,1)` даст новый объект `Vec2` с координатами $x = 3{,}0$ и $y = 4{,}0$. Однако полученный результат трудно будет прочитать в интерактивном сеансе, потому что Python не предусматривает особого форматирования значений `Vec2`:

```
>>> 3.0 * Vec2(1,0) + 4.0 * Vec2(0,1)
<__main__.Vec2 at 0x1cef56d6390>
```

Интерпретатор Python вывел адрес объекта `Vec2` в памяти, но мы уже знаем, что эта информация не важна для нас. К счастью, мы можем повлиять на преобразование экземпляров `Vec2` в строку, переопределив метод `__repr__`:

```
class Vec2():
    ...
    def __repr__(self):
        return "Vec2({},{})".format(self.x,self.y)
```

В таком строковом представлении более четко видны координаты — самые важные данные для объектов `Vec2`. Теперь результаты арифметических операций ясно видны:

```
>>> 3.0 * Vec2(1,0) + 4.0 * Vec2(0,1)
Vec2(3.0,4.0)
```

Здесь выполняются те же математические операции, которые мы выполняли с векторами-кортежами, но, как мне кажется, новая форма записи куда нагляднее. Создание класса потребовало написания некоторого шаблонного кода, например, метода определения равенства, но оно также позволило использовать прием перегрузки операторов для реализации векторной арифметики. Новый метод получения строкового представления позволяет понять, что мы работаем не просто с произвольными кортежами, но с двухмерными векторами, которые предполагают определенный способ обращения с ними. Теперь можно перейти к реализации трехмерных векторов, представленных отдельным классом.

6.1.3. Повторение процесса для трехмерных векторов

Класс, представляющий трехмерные векторы, я назову Vec3. Он очень похож на класс двухмерных векторов Vec2, только вместо двух координат будет хранить три координаты. В каждом методе Vec3, явно ссылающемся на координаты, важно правильно использовать значения x, y и z:

```
class Vec3():
    def __init__(self,x,y,z): #1
        self.x = x
        self.y = y
        self.z = z
    def add(self, other):
        return Vec3(self.x + other.x, self.y + other.y, self.z + other.z)
    def scale(self, scalar):
        return Vec3(scalar * self.x, scalar * self.y, scalar * self.z)
    def __eq__(self,other):
        return (self.x == other.x
                and self.y == other.y
                and self.z == other.z)
    def __add__(self, other):
        return self.add(other)
    def __mul__(self, scalar):
        return self.scale(scalar)
    def __rmul__(self,scalar):
        return self.scale(scalar)
    def __repr__(self):
        return "Vec3({},{},{})".format(self.x,self.y, self.z)
```

Теперь можно попробовать написать на Python код, выполняющий арифметические действия с использованием встроенных операторов:

```
>>> 2.0 * (Vec3(1,0,0) + Vec3(0,1,0))
Vec3(2.0,2.0,0.0)
```

Класс Vec3, как и класс Vec2, дает хорошую возможность подумать об обобщении. Есть несколько направлений, в которых мы можем пойти, и в данном случае, как и при проектировании любого другого программного обеспечения, выбор во многом определяется личными предпочтениями. Можно, например, сосредоточиться на упрощении арифметики: вместо реализации двух разных методов add для Vec2 и Vec3 использовать функцию add, написанную в главе 3, которая способна обрабатывать векторы координат любой размерности. Можно также реализовать хранение координаты внутри класса в виде кортежа или списка и позволить конструктору принимать любое количество координат и создавать векторы любой размерности. Однако я оставлю реализацию этих возможностей вам в качестве упражнения и поведу вас в другом направлении.

Обобщение, на котором я хочу сосредоточиться, касается использования векторов, а не особенностей их работы. Это приводит нас к мысленной модели,

которая хорошо организует код и соответствует математическому определению вектора. Например, можно написать обобщенную функцию `average` вычисления среднего вектора — координат середины отрезка, соединяющего два вектора любого типа:

```
def average(v1,v2):
    return 0.5 * v1 + 0.5 * v2
```

Этой функции можно передать и двухмерные, и трехмерные векторы, например, оба вызова — `average(Vec2 (9.0, 1.0), Vec2 (8.0, 6.0))` и `average(Vec3 (1,2,3), Vec3 (4,5,6))` — дадут верные и значимые результаты. С ее помощью можно, к примеру, усреднять изображения. Реализовав подходящий класс для представления изображений, мы сможем написать `average(img1, img2)` и получить новое изображение.

В этом проявляется красота и экономия обобщения. Мы можем написать одну универсальную функцию, такую как `average`, и применять ее к самым разным типам векторов. Единственное ограничение — входные векторы должны поддерживать умножение на скаляр и векторное сложение. Конкретные реализации арифметических операций могут различаться для `Vec2`, `Vec3`, изображений или других типов данных, но все эти типы должны совпадать в том, *что* с ними можно делать. Отделяя *что* от *как*, мы открываем путь к повторному применению кода и далеко идущим математическим утверждениям.

Как лучше всего описать, *что* можно делать с векторами, не касаясь конкретных деталей реализации? В Python для этого можно использовать абстрактный базовый класс.

6.1.4. Конструирование базового класса векторов

К числу основных действий, которые можно выполнять с `Vec2` и `Vec3`, относятся создание нового экземпляра, сложение с другим вектором, умножение на скаляр, проверка равенства с другим вектором и представление экземпляра в виде строки. Из них только сложение и умножение на скаляр — специфические векторные операции. Любой новый класс в Python автоматически включает остальные. Это приводит нас к следующему определению базового класса `Vector`:

```
from abc import ABCMeta, abstractmethod

class Vector(metaclass=ABCMeta):
    @abstractmethod
    def scale(self,scalar):
        pass
    @abstractmethod
    def add(self,other):
        pass
```

Модуль `abc` содержит вспомогательные классы, функции и декораторы методов, помогающие определить абстрактный базовый класс, не предназначенный для создания экземпляров. Цель абстрактного класса — служить шаблоном для других классов, наследующих его. Декоратор `@abstractmethod` означает, что метод не реализован в базовом классе и обязательно должен быть реализован в любом дочернем классе. Например, если вы попытаетесь создать экземпляр `Vector` так: `v = Vector()`, то в ответ получите следующую ошибку `TypeError`: TypeError: Can't instantiate abstract class Vector with abstract methods add, scale (TypeError: нельзя создать экземпляр абстрактного класса Vector с абстрактными методами add, scale).

В таком поведении абстрактных классов есть определенный смысл — в природе нет «просто вектора». Он должен иметь какое-то конкретное воплощение, например, в виде списка координат, стрелки на плоскости или чего-то еще. И все же этот базовый класс небесполезен, потому что заставляет любой дочерний класс реализовать необходимые методы. Полезно иметь такой базовый класс еще и потому, что его можно снабдить всеми методами, зависящими только от сложения и умножения на скаляр, такими как перегруженные операторы умножения и сложения:

```
class Vector(metaclass=ABCMeta):
    ...
    def __mul__(self, scalar):
        return self.scale(scalar)
    def __rmul__(self, scalar):
        return self.scale(scalar)
    def __add__(self,other):
        return self.add(other)
```

В отличие от абстрактных методов `scale` и `add`, эти реализации становятся автоматически доступными в любом дочернем классе. Теперь можно упростить `Vec2` и `Vec3`, унаследовав в них базовый класс `Vector`. Вот новая реализация `Vec2`:

```
class Vec2(Vector):
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def add(self,other):
        return Vec2(self.x + other.x, self.y + other.y)
    def scale(self,scalar):
        return Vec2(scalar * self.x, scalar * self.y)
    def __eq__(self,other):
        return self.x == other.x and self.y == other.y
    def __repr__(self):
        return "Vec2({},{})".format(self.x, self.y)
```

Этот прием действительно избавил нас от повторяющегося кода! Методы, одинаковые в классах `Vec2` и `Vec3`, теперь находятся в классе `Vector`. Все остальные методы, присутствующие в `Vec2`, специфичны для двухмерных векторов, их

нужно модифицировать так, чтобы они работали и с `Vec3` (вам будет предложено сделать это в упражнениях), и с векторами любых других размерностей.

Базовый класс `Vector` хорошо представляет возможные действия с векторами. Если мы придумаем еще какие-нибудь полезные методы, которые можно было бы в него добавить, то, скорее всего, они будут полезны для векторов любых типов. Например, в `Vector` можно добавить два таких метода:

```
class Vector(metaclass=ABCMeta):
    ...
    def subtract(self,other):
        return self.add(-1 * other)
    def __sub__(self,other):
        return self.subtract(other)
```

И тогда без внесения каких-либо изменений в `Vec2` мы автоматически получим возможность вычитать векторы:

```
>>> Vec2(1,3) - Vec2(5,1)
Vec2(-4,2)
```

Этот абстрактный класс упрощает реализацию общих векторных операций и одновременно согласуется с математическим определением вектора. Теперь переключимся с языка Python на простой человеческий и посмотрим, как абстракция переносится из кода на настоящее математическое определение.

6.1.5. Определение векторных пространств

В математике вектор определяется тем, что он делает, а не тем, чем он является. Примерно так мы и определили абстрактный класс `Vector`. Вот первое (неполное) определение вектора.

> Вектор — это объект, *позволяющий* складывать его с другими векторами и умножать на скаляры.

Наши объекты `Vec2` и `Vec3`, как и любые другие объекты, наследующие класс `Vector`, можно складывать друг с другом и умножать на скаляры. Это определение неполное, потому что я не сказал, что означает «позволяет», а это, как оказывается, самая важная часть определения!

Есть несколько важных правил, запрещающих нежелательное поведение, о многих из которых вы наверняка уже догадались. Нет необходимости запоминать все эти правила. Но если вам когда-нибудь понадобится проверить, подходит ли новый тип объекта под определение вектора, то вы можете вернуться к ним. Первый набор правил говорит о том, что сложение должно выполняться корректно.

1. Порядок следования векторов в операции сложения не имеет значения: $\mathbf{v} + \mathbf{w} = \mathbf{w} + \mathbf{v}$ для любых векторов $\mathbf{v}$ и $\mathbf{w}$.

2. Группировка векторов в операции сложения не должна иметь значения: выражение $\mathbf{u} + (\mathbf{v} + \mathbf{w})$ должно давать такой же результат, как и выражение $(\mathbf{u} + \mathbf{v}) + \mathbf{w}$, а это означает, что выражение вида $\mathbf{u} + \mathbf{v} + \mathbf{w}$ должно определяться однозначно.

Хороший контрпример — сложение строк путем конкатенации. В Python можно вычислить сумму `"hot" + "dog"`, но такое суммирование не соответствует случаю сложения векторов, потому что суммы `"hot" + "dog"` и `"dog" + "hot"` не равны и нарушают правило 1.

Умножение на скаляр тоже должно давать корректный результат и быть совместимо со сложением. Например, умножение на целое число должно давать тот же результат, что и многократное сложение (например, $3\mathbf{v} = \mathbf{v} + \mathbf{v} + \mathbf{v}$). Вот конкретные правила.

1. Умножение векторов на несколько скаляров должно давать тот же результат, что и умножение вектора на произведение скаляров. Если a и b — скаляры, а $\mathbf{v}$ — вектор, то результат $a \cdot (b \cdot \mathbf{v})$ должен совпадать с результатом $(a \cdot b) \cdot \mathbf{v}$.
2. Умножение вектора на 1 не должно изменять его: $1 \cdot \mathbf{v} = \mathbf{v}$.
3. Сумма произведений вектора на скаляры должна быть равна произведению вектора на сумму скаляров: результат $a \cdot \mathbf{v} + b \cdot \mathbf{v}$ должен совпадать с результатом $(a + b) \cdot \mathbf{v}$.
4. Произведение суммы векторов на скаляр должно быть равно сумме произведений векторов на скаляр: результат $a \cdot (\mathbf{v} + \mathbf{w})$ должен совпадать с результатом $a \cdot \mathbf{v} + a \cdot \mathbf{w}$.

Ни одно из этих правил не должно вызывать удивления. Например, $3 \cdot \mathbf{v} + 5 \cdot \mathbf{v}$ можно перевести на простой язык как «вектор **v**, увеличенный в 3 раза, плюс вектор **v**, увеличенный в 5 раз». Конечно, это то же самое, что «вектор **v**, увеличенный в 8 раз», или $8 \cdot \mathbf{v}$ согласно правилу 5.

Из этих правил следует, что не все операции сложения и умножения одинаковы. Мы должны проверить каждое правило и убедиться, что сложение и умножение выполняются корректно. Если правила соблюдаются, то рассматриваемые объекты можно с полным основанием назвать векторами.

Векторное пространство — это совокупность совместимых векторов. Вот его определение.

> Векторное пространство — это совокупность объектов, называемых векторами, которые поддерживают операции сложения векторов и умножения на скаляр (в соответствии с правилами, приведенными ранее) так, что каждая линейная комбинация векторов в совокупности дает вектор, который также относится к этой совокупности.

Набор, такой как `[Vec2(1,0), Vec2(5,-3), Vec2(1.1,0.8)]`, — это группа векторов, которые можно складывать и перемножать, но это не векторное пространство. Например, `1 * Vec2(1,0) + 1 * Vec2(5,-3)` — это линейная комбинация, результатом которой является вектор `Vec2(6,-3)`, отсутствующий в наборе. Одним из примеров векторного пространства может служить бесконечный набор всех возможных двухмерных векторов. На самом деле большинство векторных пространств, с которыми вы можете столкнуться, представляют собой бесконечные множества. В конце концов, существует бесконечно много линейных комбинаций, использующих бесконечно много скаляров!

Из того факта, что векторные пространства должны содержать все скалярные множители, вытекают два следствия, достаточно важных для того, чтобы упомянуть их отдельно. Во-первых, для любого вектора **v** в векторном пространстве выражение $0 \cdot \mathbf{v}$ дает один и тот же результат, который называется *нулевым вектором* и обозначается **0** (здесь ноль выделен жирным, чтобы отличить его от числа 0). Сложение нулевого вектора с любым другим вектором дает в результате этот другой вектор: $\mathbf{0} + \mathbf{v} = \mathbf{v} + \mathbf{0} = \mathbf{v}$. Второе следствие состоит в том, что для каждого вектора **v** имеется противоположный вектор $-1 \cdot \mathbf{v}$, который можно записать как $-\mathbf{v}$. Согласно правилу $5\ \mathbf{v} + -\mathbf{v} = (1 + -1) \cdot \mathbf{v} = 0 \cdot \mathbf{v} = \mathbf{0}$. Для каждого вектора в векторном пространстве имеется другой вектор, который компенсирует его сложением. В качестве упражнения можете попробовать усовершенствовать класс `Vector`, добавив нулевой вектор и функцию отрицания как обязательные члены.

Такой класс, как `Vec2` или `Vec3`, сам по себе не набор, но он описывает набор значений. То есть мы можем рассматривать классы `Vec2` и `Vec3` как два разных векторных пространства, а их экземпляры — как отдельные векторы. В следующем разделе увидим гораздо больше примеров векторных пространств с представляющими их классами, но сначала посмотрим, как проверить соответствие конкретным правилам, которые мы разобрали.

6.1.6. Модульное тестирование классов векторных пространств

До сих пор было удобно рассматривать абстрактный класс `Vector` как образец того, что должен реализовать вектор, а не того, как он это реализует. Но простое наличие в базовом классе того или иного абстрактного метода, такого как `add`, не гарантирует, что каждый дочерний класс реализует соответствующую операцию согласно установленным правилам.

В математике можно гарантировать соответствие правилам, *написав доказательство*. В программном коде, особенно на таком динамическом языке, как Python, лучшее, что можно сделать, — написать модульные тесты. Например, чтобы

проверить соответствие правилу 6 из предыдущего раздела, можно создать два вектора и скаляр и проверить выполнение равенства:

```
>>> s = -3
>>> u, v = Vec2(42,-10), Vec2(1.5, 8)
>>> s * (u + v) == s * v + s * u
True
```

Часто именно так записываются модульные тесты, но это довольно слабый тест, потому что он опробует только один пример. Его можно сделать сильнее, использовав случайные числа. В следующем примере я использую функцию `random.uniform`, чтобы сгенерировать равномерно распределенные числа с плавающей точкой в диапазоне от –10 до 10:

```
from random import uniform

def random_scalar():
    return uniform(-10,10)

def random_vec2():
    return Vec2(random_scalar(),random_scalar())

a = random_scalar()
u, v = random_vec2(), random_vec2()
assert a * (u + v) == a * v + a * u
```

Если вам не повезет, этот тест завершится ошибкой `AssertionError`. Вот значения a, u и v, с которыми мой тест потерпел неудачу:

```
>>> a, u, v
(0.17952747449930084,
Vec2(0.8353326458605844,0.2632539730989293),
Vec2(0.555146137477196,0.34288853317521084))
```

Выражения слева и справа от оператора сравнения в вызове `assert` из предыдущего примера дают такие значения:

```
>>> a * (u + v), a * u + a * v
(Vec2(0.24962914431749222,0.10881923333807299),
Vec2(0.24962914431749225,0.108819233338073))
```

Да, эти векторы разные, но только потому, что их компоненты различаются на несколько квадриллионных долей (очень и очень маленькие значения). Это не означает, что математика неверна, просто арифметика с плавающей точкой имеет ограниченную точность.

Чтобы игнорировать такие небольшие несоответствия, можно использовать для проверки другое понятие равенства. Функция `math.isclose` проверяет близость двух значений с плавающей точкой в пределах заданной погрешности

(по умолчанию близкими считаются значения, отличающиеся не более чем на одну миллиардную долю от большего значения). Если задействовать ее для проверки, тест успешно проходит 100 раз подряд:

```
from math import isclose

def approx_equal_vec2(v,w):
    return isclose(v.x,w.x) and isclose(v.y,w.y)

for _ in range(0,100):
    a = random_scalar()
    u, v = random_vec2(), random_vec2()
    assert approx_equal_vec2(a * (u + v),
                             a * v + a * u)
```

Проверить близость компонентов x и y (они необязательно должны быть равны)

Проверить 100 разных случайных скаляров и пар векторов

Вместо строгого равенства проверить близость векторов с помощью новой функции

Удалив из уравнения ошибку, обусловленную ограниченной точностью вычислений с плавающей точкой, можно протестировать все шесть свойств векторного пространства:

```
def test(eq, a, b, u, v, w):
    assert eq(u + v, v + u)
    assert eq(u + (v + w), (u + v) + w)
    assert eq(a * (b * v), (a * b) * v)
    assert eq(1 * v, v)
    assert eq((a + b) * v, a * v + b * v)
    assert eq(a * v + a * w, a * (v + w))

for i in range(0,100):
    a,b = random_scalar(), random_scalar()
        u,v,w = random_vec2(), random_vec2(), random_vec2()
        test(approx_equal_vec2,a,b,u,v,w)
```

Выполняет проверку равенства, вызывая функцию eq. Это делает функцию test независимой от конкретной реализации вектора

Этот тест показывает, что все шесть правил (свойств) выполняются для 100 различных случайно выбранных скаляров и векторов. Успешное прохождение 600 рандомизированных модульных тестов — убедительный признак того, что класс `Vec2` соответствует правилам, перечисленным в предыдущем разделе. Реализовав в упражнениях свойство `zero()` и оператор отрицания, вы сможете добавить тестирование еще нескольких свойств.

Такой способ тестирования не универсален — нам пришлось написать специальные функции, генерирующие случайные экземпляры `Vec2` и сравнивающие их. Но сама функция `test` и выражения внутри нее абсолютно универсальны. Пока тестируемый класс наследует `Vector`, он сможет участвовать в таких выражениях, как `a * v + a * w` и `a * (v + w)`, которые затем можно проверить на равенство (близость). Теперь мы можем погрузиться в изучение различных объектов, которые можно рассматривать как векторы, и мы знаем, как их протестировать, если понадобится.

6.1.7. Упражнения

Упражнение 6.1. Реализуйте класс `Vec3`, наследующий `Vector`.

Решение

```
class Vec3(Vector):
    def __init__(self,x,y,z):
        self.x = x
        self.y = y
        self.z = z
    def add(self,other):
        return Vec3(self.x + other.x,
                    self.y + other.y,
                    self.z + other.z)
    def scale(self,scalar):
        return Vec3(scalar * self.x,
                    scalar * self.y,
                    scalar * self.z)
    def __eq__(self,other):
        return (self.x == other.x
                and self.y == other.y
                and self.z == other.z)
    def __repr__(self):
        return "Vec3({},{},{})".format(self.x, self.y, self.z)
```

Упражнение 6.2. Мини-проект. Реализуйте класс `CoordinateVector`, наследующий `Vector`, с абстрактным свойством, представляющим мерность. Это должно избавить от повторной работы при реализации определенных классов векторов координат. Наследования `CoordinateVector` и установки свойства мерности равным `6` должно быть достаточно для реализации класса `Vec6`.

Решение. Мы можем использовать не зависящие от мерности операции сложения и умножения на скаляр из глав 2 и 3. Единственный нереализованный метод в следующем классе — `dimension`, и это не позволяет создать экземпляр `CoordinateVector`:

```
from abc import abstractproperty
from vectors import add, scale

class CoordinateVector(Vector):
    @abstractproperty
    def dimension(self):
        pass
    def __init__(self,*coordinates):
        self.coordinates = tuple(x for x in coordinates)
    def add(self,other):
        return self.__class__(*add(self.coordinates, other.coordinates))
```

```
    def scale(self,scalar):
        return self.__class__(*scale(scalar, self.coordinates))
    def __repr__(self):
        return "{}{}".format(self.__class__.__qualname__, self.coordinates)
```

После выбора мерности (например, 6) можно определить конкретный класс, экземпляры которого можно будет создавать:

```
class Vec6(CoordinateVector):
    def dimension(self):
        return 6
```

Определения методов сложения, умножения на скаляр и других будут наследоваться от базового класса `CoordinateVector`:

```
>>> Vec6(1,2,3,4,5,6) + Vec6(1, 2, 3, 4, 5, 6)
Vec6(2, 4, 6, 8, 10, 12)
```

Упражнение 6.3. Добавьте в класс `Vector` абстрактный метод `zero`, возвращающий нулевой вектор в данном векторном пространстве, а также реализацию оператора отрицания. Часто бывает полезно иметь возможность получить нулевой вектор и вычислить отрицание любого вектора в векторном пространстве.

Решение

```
from abc import ABCMeta, abstractmethod, abstractproperty

class Vector(metaclass=ABCMeta):
    ...
    @classmethod
    @abstractproperty
    def zero():
        pass

    def __neg__(self):
        return self.scale(-1)
```

zero — это метод класса, потому что в любом векторном пространстве есть только один нулевой вектор

Кроме того, это абстрактное свойство, потому что мы пока не знаем, что такое нулевой вектор

Специальное имя метода, соответствующее оператору отрицания

Реализовывать свои версии метода `__neg__` в дочерних классах не потребуется, потому что он уже реализован в родительском классе и основан только на умножении на скаляр. Но метод `zero` придется реализовать в каждом дочернем классе:

```
class Vec2(Vector):
    ...
    def zero():
        return Vec2(0,0)
```

Упражнение 6.4. Напишите модульные тесты, показывающие, что операции сложения векторов и умножения вектора на скаляр в `Vec3` удовлетворяют свойствам векторного пространства.

Решение. Поскольку функция `test` универсальна, достаточно написать новые функции проверки равенства и получения случайных координат для объектов `Vec3`:

```
def random_vec3():
    return Vec3(random_scalar(),random_scalar(),random_scalar())

def approx_equal_vec3(v,w):
    return isclose(v.x,w.x) and isclose(v.y,w.y) and isclose(v.z, w.z)

for i in range(0,100):
    a,b = random_scalar(), random_scalar()
    u,v,w = random_vec3(), random_vec3(), random_vec3()
    test(approx_equal_vec3,a,b,u,v,w)
```

Упражнение 6.5. Добавьте модульные тесты, проверяющие выполнение условий $\mathbf{0} + \mathbf{v} = \mathbf{v}$, $0 \cdot \mathbf{v} = \mathbf{0}$ и $-\mathbf{v} + \mathbf{v} = \mathbf{0}$ для любого вектора $\mathbf{v}$, где 0 — это число ноль, а $\mathbf{0}$ — нулевой вектор.

Решение. Поскольку нулевой вектор зависит от тестируемого класса, он должен передаваться в функцию как аргумент:

```
def test(zero,eq,a,b,u,v,w):
    ...
    assert eq(zero + v, v)
    assert eq(0 * v, zero)
    assert eq(-v + v, zero)
```

Вот как можно протестировать любой класс векторов с реализованным методом `zero` (см. упражнение 6.3):

```
for i in range(0,100):
    a,b = random_scalar(), random_scalar()
    u,v,w = random_vec2(), random_vec2(), random_vec2()
    test(Vec2.zero(), approx_equal_vec2, a,b,u,v,w)
```

Упражнение 6.6. Поскольку для `Vec2` и `Vec3` реализованы свои методы проверки равенства, выражение `Vec2(1,2) == Vec3(1,2,3)` дает `True`. «Утиная» типизация в языке Python слишком снисходительна! Исправьте эту ошибку, добавив проверку соответствия классов сравниваемых векторов.

Решение. Как оказалось, аналогичную проверку надо еще и в реализацию сложения добавить!

```
class Vec2(Vector):
    ...
    def add(self,other):
        assert self.__class__ == other.__class__
        return Vec2(self.x + other.x, self.y + other.y)

    ...
    def __eq__(self,other):
        return (self.__class__ == other.__class__
            and self.x == other.x and self.y == other.y)
```

На всякий случай такие же проверки можно добавить в другие дочерние классы `Vector`.

Упражнение 6.7. Реализуйте функцию `__truediv__` для `Vector`, которая позволяет делить векторы на скаляры. Чтобы разделить вектор на ненулевой скаляр, его можно просто умножить на величину, обратную скаляру (1 / скаляр).

Решение.

```
class Vector(metaclass=ABCMeta):
    ...
    def __truediv__(self, scalar):
        return self.scale(1.0/scalar)
```

С помощью этого метода можно, например, выполнить деление `Vec2(1, 2)/2` и получить в результате `Vec2(0.5, 1.0)`.

6.2. ИССЛЕДОВАНИЕ РАЗЛИЧНЫХ ВЕКТОРНЫХ ПРОСТРАНСТВ

Теперь, получив представление о векторных пространствах, рассмотрим несколько примеров. В каждом из них реализуем новый тип объектов как класс, наследующий `Vector`, и покажем, что независимо от конкретного типа с его экземплярами можно выполнять сложение, умножение на скаляр и любые другие векторные операции.

6.2.1. Перечисление всех пространств координатных векторов

До сих пор мы много внимания уделяли векторам координат `Vec2` и `Vec3`, поэтому двух- и трехмерные векторы не нуждаются в дополнительных пояснениях. Однако повторю еще раз, что пространство координатных векторов может иметь любое количество измерений. Векторы `Vec2` имеют два измерения, векторы `Vec3` — три, и мы могли бы точно так же определить класс `Vec15`, представляющий 15-мерное векторное пространство. Объекты `Vec15` нельзя изобразить геометрически, потому что они представляют точки в 15-мерном пространстве.

Стоит упомянуть один особый случай — класс, который можно назвать `Vec1`, представляющий векторы с одной координатой. Вот как выглядит его реализация:

```
class Vec1(Vector):
    def __init__(self,x):
        self.x = x
    def add(self,other):
        return Vec1(self.x + other.x)
    def scale(self,scalar):
        return Vec1(scalar * self.x)
    @classmethod
    def zero(cls):
        return Vec1(0)
    def __eq__(self,other):
        return self.x == other.x
    def __repr__(self):
        return "Vec1({})".format(self.x)
```

Как видите, определение класса содержит слишком много шаблонного кода для простого представления единственного числа и не содержит никакой арифметики, которой бы у нас не было. Сложение и умножение на скаляр объектов `Vec1` — это простое сложение и умножение базовых чисел:

```
>>> Vec1(2) + Vec1(2)
Vec1(4)
>>> 3 * Vec1(1)
Vec1(3)
```

По этой причине трудно придумать ситуацию, когда мог бы пригодиться класс `Vec1`. И тем не менее важно понимать, что числа сами по себе являются векторами. Совокупность всех действительных чисел, включая целые, дробные и иррациональные числа, такие как π, обозначается $\mathbb{R}$ и является полноценным векторным пространством. Это особый случай, когда скаляры и векторы являются объектами одного типа.

Пространство координатных векторов обозначается $\mathbb{R}^n$, где n — размерность, или количество координат. Например, двухмерная плоскость обозначается $\mathbb{R}^2$, а трехмерное пространство — $\mathbb{R}^3$. При условии использования действительных чисел в качестве скаляров любое векторное пространство представляет собой некоторое замаскированное $\mathbb{R}^n$. (То есть пока гарантируется конечное число измерений в векторном пространстве! Существует векторное пространство, обозначаемое $\mathbb{R}^\infty$, но это не единственное бесконечномерное векторное пространство.) Вот почему мы должны помнить о векторном пространстве $\mathbb{R}$, даже притом что оно весьма примитивно. Другое векторное пространство, о котором следует упомянуть, — *нульмерное пространство* $\mathbb{R}^0$. Это совокупность векторов с нулевым количеством координат, которые можно описать как пустые кортежи или как класс `Vec0`, наследующий `Vector`:

```
class Vec0(Vector):
    def __init__(self):
        pass
    def add(self,other):
        return Vec0()
    def scale(self,scalar):
        return Vec0()
    @classmethod
    def zero(cls):
        return Vec0()
    def __eq__(self,other):
        return self.__class__ == other.__class__ == Vec0
    def __repr__(self):
        return "Vec0()"
```

Отсутствие координат не означает отсутствие возможных векторов — это означает, что существует ровно один нульмерный вектор. Это делает нульмерную векторную математику невероятно простой — все операции дают один и тот же результат:

```
>>> -3.14 * Vec0()
Vec0()
>>> Vec0() + Vec0() + Vec0() + Vec0()
Vec0()
```

С точки зрения ООП это что-то вроде класса-синглтона[1]. С математической точки зрения каждое векторное пространство должно иметь нулевой вектор, поэтому `Vec0()` можно считать таким нулевым вектором.

[1] Класса, имеющего единственный экземпляр. — *Примеч. пер.*

Все сказанное ранее относится к векторам координат любых мерностей: ноль, один, два, три или более. Теперь, встретившись с вектором в «дикой природе», вы сможете сопоставить его с одним из этих векторных пространств.

6.2.2. Идентификация векторных пространств в «дикой природе»

Вернемся к примеру из главы 1 и рассмотрим набор данных с информацией о подержанных автомобилях Toyota Prius. В исходном коде, прилагаемом к книге, вы увидите, как загрузить набор данных, любезно предоставленный моим другом Дэном Рэтбоуном (Dan Rathbone) из CarGraph.com. Для простоты я загружаю эти данные в класс:

```
class CarForSale():
    def __init__(self, model_year, mileage, price, posted_datetime,
                 model, source, location, description):
        self.model_year = model_year
        self.mileage = mileage
        self.price = price
        self.posted_datetime = posted_datetime
        self.model = model
        self.source = source
        self.location = location
        self.description = description
```

Было бы полезно представить объекты `CarForSale` как векторы. Тогда, например, можно было бы усреднить их как линейную комбинацию и посмотреть, как выглядит типичный Prius, выставленный на продажу. Для этого нужно модифицировать класс `CarForSale` так, чтобы он наследовал `Vector`.

Как выполнить сложение двух машин? Числовые поля, такие как `model_year`, `mileage` и `price`, можно сложить как компоненты вектора, но трудно представить более или менее осмысленную интерпретацию сложения строковых полей. (Напомню, что мы не можем манипулировать строками так же, как векторами.) В результате арифметического действия с автомобилями получается не реальный автомобиль, выставленный на продажу, а некий *виртуальный* автомобиль, определяемый его свойствами. А чтобы напомнить об этом, я запишу во все строковые свойства результата строку `"(virtual)"`. Наконец, мы не можем складывать дату и время, зато можем складывать промежутки времени. На рис. 6.3 я использую день, когда были получены данные, как точку отсчета и складываю промежутки времени, прошедшие с момента выставления автомобиля на продажу до этого дня. В листинге 6.1 приводится полный код измененного класса.

То же верно в отношении умножения на скаляр. Мы можем умножить на скаляр числовые свойства и интервал времени, прошедшего с момента публикации объявления, однако строковые свойства при этом теряют смысл.

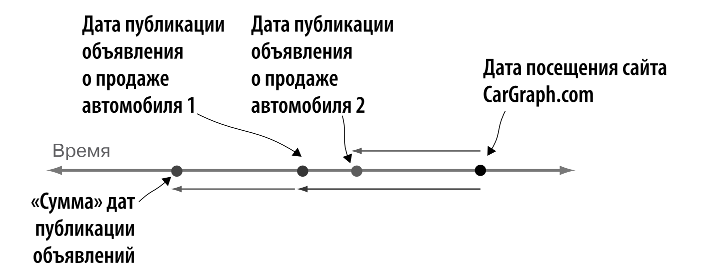


Рис. 6.3. Хронология выставления автомобилей на продажу

Листинг 6.1. Преобразование класса CarForSale в вектор с реализацией необходимых методов

```
from datetime import datetime

class CarForSale(Vector):
    retrieved_date = datetime(2018,11,30,12)
    def __init__(self, model_year, mileage, price, posted_datetime,
                 model="(virtual)", source="(virtual)",
                 location="(virtual)", description="(virtual)"):
        self.model_year = model_year
        self.mileage = mileage
        self.price = price
        self.posted_datetime = posted_datetime
        self.model = model
        self.source = source
        self.location = location
        self.description = description

    def add(self, other):
        def add_dates(d1, d2):
            age1 = CarForSale.retrieved_date - d1
            age2 = CarForSale.retrieved_date - d2
            sum_age = age1 + age2
            return CarForSale.retrieved_date - sum_age
        return CarForSale(
            self.model_year + other.model_year,
            self.mileage + other.mileage,
            self.price + other.price,
            add_dates(self.posted_datetime, other.posted_datetime)
        )

    def scale(self,scalar):
        def scale_date(d):
            age = CarForSale.retrieved_date - d
            return CarForSale.retrieved_date - (scalar * age)
        return CarForSale(
            scalar * self.model_year,
            scalar * self.mileage,
```

Я получил набор данных с CarGraph.com 30.11.2018 в полдень

Для простоты создания виртуальных автомобилей все строковые параметры считаются необязательными и по умолчанию им присваивается значение "(virtual)"

Вспомогательная функция, складывающая даты путем сложения интервалов до точки отсчета

Складывает объекты CarForSale путем сложения значений их свойств и создает новый объект

Вспомогательная функция для умножения даты/времени на скаляр как интервала времени от точки отсчета

```
            scalar * self.price,
            scale_date(self.posted_datetime)
        )

    @classmethod
    def zero(cls):
        return CarForSale(0, 0, 0, CarForSale.retrieved_date)
```

Полную реализацию класса, а также код, загружающий список образцов автомобилей, вы найдете в примерах, сопровождающих книгу. Загрузив список автомобилей, можно попробовать выполнить некоторые векторные операции:

```
>>> (cars[0] + cars[1]).__dict__
{'model_year': 4012,
 'mileage': 306000.0,
 'price': 6100.0,
 'posted_datetime': datetime.datetime(2018, 11, 30, 3, 59),
 'model': '(virtual)',
 'source': '(virtual)',
 'location': '(virtual)',
 'description': '(virtual)'}
```

Сумма первых двух автомобилей, очевидно, представляет Prius модельного года 4012 (может быть, он способен летать?) с пробегом 306 000 миль и запрашиваемой ценой 6100 долларов. Он был выставлен на продажу в 3:59 в тот же день, когда я заглянул на CarGraph.com. Этот необычный виртуальный автомобиль выглядит бесполезным, но давайте взглянем на средние значения (показаны далее), которые выглядят намного более осмысленными:

```
>>> average_prius = sum(cars, CarForSale.zero()) * (1.0/len(cars))
>>> average_prius.__dict__

{'model_year': 2012.5365853658536,
 'mileage': 87731.63414634147,
 'price': 12574.731707317074,
 'posted_datetime': datetime.datetime(2018, 11, 30, 9, 0, 49, 756098),
 'model': '(virtual)',
 'source': '(virtual)',
 'location': '(virtual)',
 'description': '(virtual)'}
```

Мы можем извлечь ценные сведения из этого результата. Среднему продаваемому автомобилю Prius около 6 лет, его пробег — около 88 000 миль, за него просят примерно 12 500 долларов, и объявление было опубликовано в 9:49, когда я зашел на сайт. (В части III книги мы потратим много времени на изучение наборов данных, рассматривая их как векторы.)

Если игнорировать текстовые данные, то можно сказать, что `CarForSale` ведет себя подобно вектору. Фактически он ведет себя как четырехмерный вектор, имеющий следующие измерения: цена, год выпуска, пробег и дата/время публикации объявления. Это не совсем координатный вектор, потому что дата публикации — это не число. Несмотря на то что не все данные числовые, этот

класс удовлетворяет свойствам векторного пространства (в упражнениях вам будет предложено проверить это с помощью модульных тестов), поэтому его экземпляры являются векторами и к ним можно применять векторные операции. В частности, это четырехмерные векторы, поэтому можно реализовать их отображение экземпляров `CarForSale` в объекты `Vec4` (это тоже будет предложено сделать в упражнениях). В следующем примере мы рассмотрим объекты, которые еще меньше похожи на координатные векторы, но все же удовлетворяют определяющим свойствам.

6.2.3. Интерпретация функций как векторов

Оказывается, математические функции можно рассматривать как векторы. В частности, я говорю о функциях, которые принимают и возвращают одно действительное число, хотя существует множество других типов математических функций. На языке математики функция f, принимающая и возвращающая действительное число, обозначается так: $f\colon \mathbb{R} \to \mathbb{R}$. На языке Python мы будем рассматривать функции, принимающие и возвращающие значения типа `float`.

Так же как в случае с двух- или трехмерными векторами, сложение и умножение функций на скаляр можно выполнять визуально или алгебраически. Для начала можно записать функцию алгебраически, например, $f(x) = 0{,}5x + 3$ или $g(x) = \sin x$. А затем визуализировать ее в виде графика.

В примерах к книге я написал простую функцию `plot`, которая рисует график одной или нескольких функций для заданного диапазона входных данных (рис. 6.4). Например, следующий код нарисует графики двух функций, $f(x)$ и $g(x)$, для значений x в интервале от -10 до 10:

```
def f(x):
    return 0.5 * x + 3
def g(x):
    return sin(x)
plot([f,g],-10,10)
```

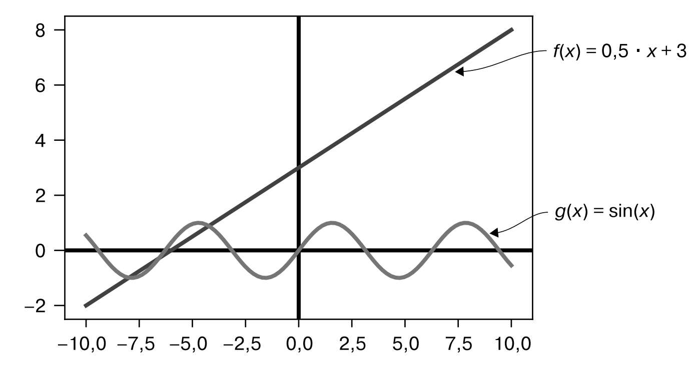


Рис. 6.4. Графики функций $f(x) = 0{,}5x + 3$ и $g(x) = \sin x$

Мы можем складывать функции алгебраически, складывая определяющие их выражения. Из этого следует, что $f + g$ — это функция, определяемая формулой $(f + g)(x) = f(x) + g(x) = 0{,}5x + 3 + \sin x$. Графически значения y каждой точки складываются, и результат похож на наложение двух функций друг на друга, как показано на рис. 6.5.

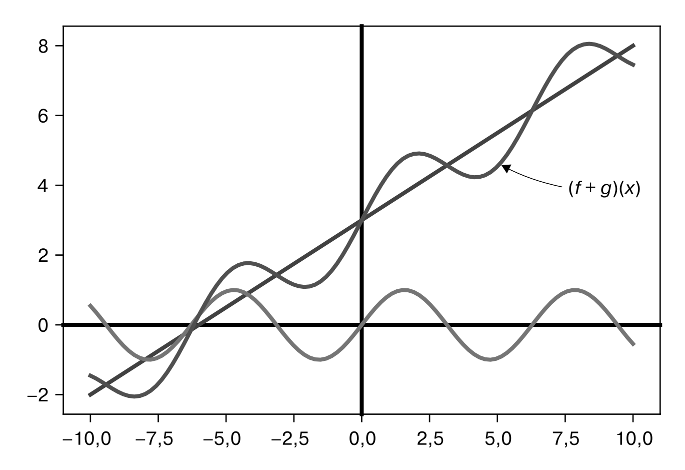


Рис. 6.5. Визуализация суммы двух функций на графике

Чтобы реализовать такое сложение, можно написать некий функциональный код на Python, принимающий две функции и возвращающий новую функцию, являющуюся их суммой:

```
def add_functions(f,g):
    def new_function(x):
        return f(x) + g(x)
    return new_function
```

Точно так же можно умножить функцию на скаляр, умножив ее выражение на скаляр. Например, $3g$ определяется как $(3g)(x) = 3 \cdot g(x) = 3 \cdot \sin x$. В данном случае это приведет к растяжению графика функции g вдоль оси y в 3 раза (рис. 6.6).

Функции можно завернуть в класс, который наследует `Vector`, что и будет предложено сделать в разделе с упражнениями. После этого вы сможете писать в коде обычные арифметические выражения с функциями, такие как $3 \cdot f$ или $2 \cdot f - 6 \cdot g$. Более того, класс функций можно даже сделать *вызываемым* и способным принимать аргументы, подобно функциям, что позволит использовать такие выражения, как $(f + g)(6)$. К сожалению, реализовать модульное тестирование для определения соответствия функций свойствам векторного пространства намного сложнее, потому что трудно сгенерировать случайные функции или проверить их равенство. Чтобы говорить о равенстве двух функций, нужно быть уверенным, что они возвращают один и тот же результат для каждого возможного входного

значения, а это означает необходимость проверки для каждого действительного числа или, по крайней мере, для каждого значения `float`!

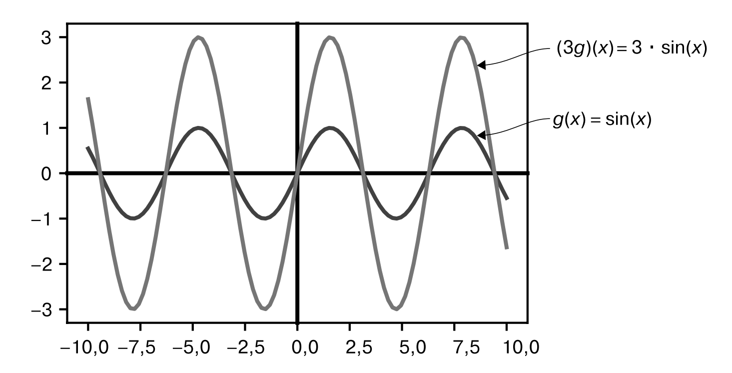


Рис. 6.6. Функция ($3g$) выглядит как функция g, растянутая в 3 раза вдоль оси y

Это подводит нас к другому вопросу — о размерности векторного пространства функций, то есть к определению количества действительных числовых координат, необходимых для однозначной идентификации функции.

Координаты x, y и z объекта `Vec3` можно обозначить не буквами, а числовыми индексами от $i = 1$ до $i = 3$. Точно так же можно проиндексировать координаты `Vec15` от $i = 1$ до $i = 15$. Однако функция определяется бесконечно большим количеством чисел, например значениями $f(x)$ для любого значения x. Иначе говоря, координаты f можно рассматривать как ее значения в каждой точке, проиндексированные всеми действительными числами, а не только несколькими первыми целыми числами. Это означает, что векторное пространство функций — *бесконечномерное*. Отсюда вытекают важные следствия, но в основном это затрудняет работу с векторным пространством всех функций. Мы вернемся к данной теме позже, предварительно рассмотрев некоторые более простые подмножества. А пока обратимся к более комфортным пространствам с конечным числом измерений и разберем еще два примера.

6.2.4. Интерпретация матриц как векторов

Матрица размером $n \times m$ — это список из $n \cdot m$ чисел. Даже притом что матрица имеет вид прямоугольной таблицы, с ней можно обращаться как с $(n \cdot m)$-мерным вектором. Единственное отличие векторного пространства, скажем, матрицы 5×3, от 15-мерного пространства координатных векторов состоит в том, что координаты представлены в виде матрицы. Но при этом мы все еще складываем и умножаем на скаляр координаты. На рис. 6.7 показано, как выглядит сложение.

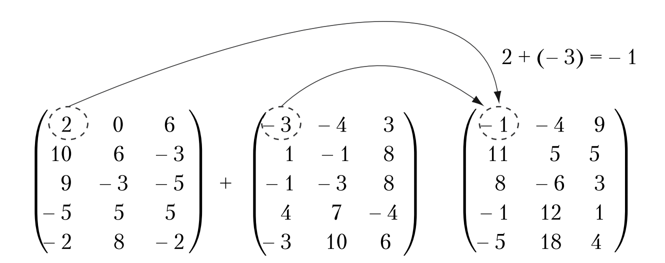


Рис. 6.7. Сложение матриц 5 × 3 путем сложения соответствующих элементов

Реализация класса матриц 5 × 3, наследующая `Vector`, требует больше кода, чем простая реализация класса `Vec15`, потому что нужно организовать два цикла для обхода элементов матрицы. Однако по своей сути арифметика ничуть не сложнее, как показано в листинге 6.2.

Листинг 6.2. Класс, представляющий матрицы 5 × 3, который можно интерпретировать как класс векторов

```
class Matrix5_by_3(Vector):
    rows = 5
    columns = 3
    def __init__(self, matrix):
        self.matrix = matrix
    def add(self, other):
        return Matrix5_by_3(tuple(
            tuple(a + b for a,b in zip(row1, row2))
            for (row1, row2) in zip(self.matrix, other.matrix)
        ))
    def scale(self,scalar):
        return Matrix5_by_3(tuple(
            tuple(scalar * x for x in row)
            for row in self.matrix
        ))
    @classmethod
    def zero(cls):
        return Matrix5_by_3(tuple(
            tuple(0 for j in range(0, cls.columns))
            for i in range(0, cls.rows)
        ))
```

Необходимо знать количество строк и столбцов, чтобы построить нулевую матрицу

Для матриц 5 × 3 нулевой вектор — это матрица 5 × 3, содержащая нули. Сложение любой другой матрицы M 5 × 3 с этой нулевой матрицей дает в результате M

С таким же успехом мы могли бы создать класс `Matrix2_by_2` или `Matrix99_by_17` для представления различных векторных пространств. Бóльшая часть реализации в таких классах останется прежней, но размерности будут уже не 15, а $2 \cdot 2 = 4$ или $99 \cdot 17 = 1683$. В качестве упражнения можете попробовать создать класс `Matrix`, наследующий `Vector`, который включает все данные, кроме количества строк и столбцов. Тогда любой класс `MatrixM_by_N` мог бы наследовать `Matrix`.

В матрицах интересно не то, что они представляют таблицы чисел, а то, что их можно рассматривать как линейные функции. Мы уже видели, что списки чисел и функций — это два случая векторных пространств, но, оказывается, матрицы являются векторами в обоих смыслах. Если матрица A имеет n строк и m столбцов, то она представляет линейную функцию, осуществляющую преобразование m-мерного пространства в n-мерное. (На математическом языке то же самое можно записать так: $A : \mathbb{R}^m \to \mathbb{R}^n$.)

Подобно тому, как мы складывали и умножали на скаляр функции из $\mathbb{R} \to \mathbb{R}$, можно складывать и умножать на скаляр функции из $\mathbb{R}^m \to \mathbb{R}^n$. В мини-проекте в конце этого раздела вам будет предложено попробовать применить модульные тесты векторного пространства к матрицам, чтобы проверить, являются ли они векторами в обоих смыслах. Это не означает, что таблицы чисел бесполезны, просто иногда бывает желательно интерпретировать их как функции. Например, массивы чисел можно использовать для представления изображений.

6.2.5. Обработка изображений с помощью векторных операций

На экране компьютера изображения выводятся в виде массивов цветных квадратиков, называемых *пикселами*. Типичное изображение может иметь по несколько сотен пикселов в высоту и ширину. В цветном изображении каждый пиксел характеризуется тремя числами (рис. 6.8), определяющими интенсивность трех базовых цветов — красного, зеленого и синего (Red, Green и Blue, RGB). Изображение размером 300 × 300 пикселов определяют 300 · 300 · 3 = 270 000 чисел. Если рассматривать изображения такого размера как векторы, то мы получим 270 000-мерное пространство!

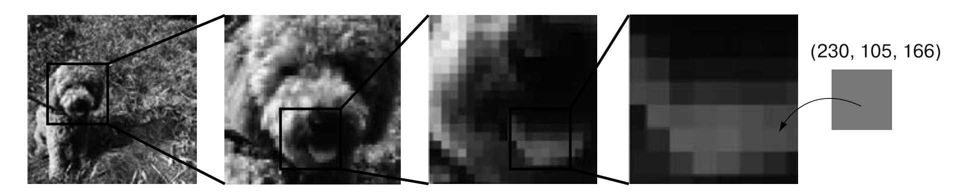


Рис. 6.8. Я последовательно увеличивал масштаб изображения моей собаки Мельбы, пока не стало возможно выделить один пиксел, характеризующийся тремя числами — интенсивностью красного, зеленого и синего цветов (230, 105, 166 соответственно)

В зависимости от способа печати книги изображение на рис. 6.8 может быть цветным или черно-белым, соответственно, вы можете не увидеть розовый цвет языка Мельбы. Но поскольку мы будем представлять цвет численно, а не визуально,

последующее обсуждение должно иметь для вас определенный смысл. А в примерах, сопровождающих книгу, вы найдете полноцветные изображения.

Для Python имеется библиотека обработки изображений PIL, ставшая стандартом де-факто. Она доступна для установки с помощью `pip` под именем `pillow`. Вам не потребуется глубоко изучать библиотеку, потому что мы сразу же инкапсулируем ее в новый класс `ImageVector` (листинг 6.3), наследующий `Vector`, хранящий пикселы изображения размером 300 × 300 и поддерживающий сложение и умножение на скаляр.

Листинг 6.3. Класс, представляющий изображение в виде вектора



```
from PIL import Image
class ImageVector(Vector):
    size = (300,300)
    def __init__(self,input):
        try:
            img = Image.open(input).\
                  resize(ImageVector.size)
            self.pixels = img.getdata()
        except:
            self.pixels = input
    def image(self):
        img = Image.new('RGB', ImageVector.size)
        img.putdata([(int(r), int(g), int(b))
                     for (r,g,b) in self.pixels])
        return img
    def add(self,img2):
        return ImageVector([(r1+r2,g1+g2,b1+b2)
                            for ((r1,g1,b1),(r2,g2,b2))
                            in zip(self.pixels,img2.pixels)])
    def scale(self,scalar):
        return ImageVector([(scalar*r,scalar*g,scalar*b)
                            for (r,g,b) in self.pixels])
    @classmethod
    def zero(cls):
        total_pixels = cls.size[0] * cls.size[1]
        return ImageVector([(0,0,0) for _ in range(0,total_pixels)])
    def _repr_png_(self):
        return self.image()._repr_png_()
```

Вооруженные этой библиотекой, мы можем загружать изображения из файлов и выполнять с ними векторные операции. Например, вот как можно вычислить среднее двух изображений и получить результат, показанный на рис. 6.9:

```
0.5 * ImageVector("inside.JPG") + 0.5 * ImageVector("outside.JPG")
```



Рис. 6.9. Среднее двух изображений Мельбы

Любой экземпляр `ImageVector` будет действительным, но минимальное и максимальное значения цветов, которые воспринимаются как визуально различающиеся, — 0 и 255 соответственно. Из-за этого отрицание любого изображения даст в результате черный квадрат, потому что каждый пиксел будет иметь яркость ниже минимальной. Аналогично, умножение на положительный скаляр будет делать изображения более бледными, так как яркость большинства пикселов превысит максимально отображаемую яркость. На рис. 6.10 показано, как это выглядит в действительности.

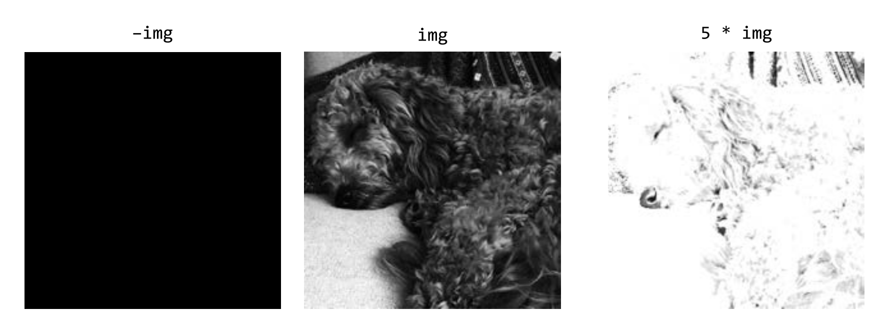


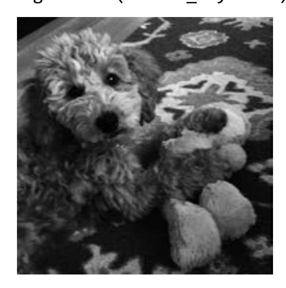
Рис. 6.10. Результат отрицания и умножения изображения на скаляр

Чтобы визуальные изменения выглядели более интересно, применяемые операции должны приводить к получению пикселов в правильном диапазоне яркостей для всех цветов. Хорошими ориентирами могут служить нулевой вектор (черный) и вектор со всеми компонентами цвета, равными 255 (белый). Например, вычитание изображения из полностью белого изображения приводит к инвертированию цветов. Как показано на рис. 6.11, вычитание изображения из следующего белого вектора:

```
white = ImageVector([(255,255,255) for _ in range(0,300*300)])
```

дает устрашающе перекрашенную картинку. (Разница впечатляет даже на черно-белом изображении.)

`ImageVector("melba_toy.JPG")`



`white - ImageVector("melba_toy.JPG")`



Рис. 6.11. Вычитание изображения из полностью белого изображения приводит к инвертированию цветов

Векторная арифметика является универсальной концепцией: определяющие понятия сложения и умножения на скаляр применимы к числам, координатным векторам, функциям, матрицам, изображениям и многим другим объектам. Поразительно видеть наглядные результаты при использовании одних и тех же математических вычислений в не связанных между собой областях. Запомним эти примеры векторных пространств и продолжим исследовать возможности их обобщения.

6.2.6. Упражнения

Упражнение 6.8. С помощью модульных тестов проверьте принадлежность к векторному пространству действительных чисел `u`, `v` и `w`, а не объектов, унаследовавших класс `Vector`. Такая проверка покажет, что действительные числа на самом деле являются векторами.

Решение. Со случайными скалярами в качестве векторов, числом 0 в качестве нулевого вектора и функцией `math.isclose` для проверки на равенство 100 проверок проходят успешно:

```
for i in range(0,100):
    a,b = random_scalar(), random_scalar()
    u,v,w = random_scalar(), random_scalar(), random_scalar()
    test(0, isclose, a,b,u,v,w)
```

Упражнение 6.9. Мини-проект. Используйте модульные тесты для проверки векторного пространства и примените их к объектам `CarForSale`, чтобы показать, что они действительно образуют векторное пространство (если игнорировать их текстовые атрибуты).

Решение. Бóльшая часть работы связана с генерацией случайных данных и реализацией проверки равенства, обрабатывающей дату и время, как показано далее:

```
from math import isclose
from random import uniform, random, randint
from datetime import datetime, timedelta

def random_time():
    return CarForSale.retrieved_date - timedelta(days=uniform(0,10))

def approx_equal_time(t1, t2):
    test = datetime.now()
    return isclose((test-t1).total_seconds(), (test-t2).total_seconds())

def random_car():
    return CarForSale(randint(1990,2019), randint(0,250000),
                      27000. * random(), random_time())

    def approx_equal_car(c1,c2):
        return (isclose(c1.model_year,c2.model_year)
                and isclose(c1.mileage,c2.mileage)
                and isclose(c1.price, c2.price)
                and approx_equal_time(c1.posted_datetime,
                                      c2.posted_datetime))

for i in range(0,100):
    a,b = random_scalar(), random_scalar()
    u,v,w = random_car(), random_car(), random_car()
    test(CarForSale.zero(), approx_equal_car, a,b,u,v,w)
```

Упражнение 6.10. Реализуйте класс `Function(Vector)`, конструктор которого принимает функцию одной переменной в аргументе, и добавьте в него метод `__call__`, чтобы экземпляры класса можно было использовать как функции. В заключение используйте экземпляры класса в вызове `plot([f, g, f+g, 3*g], -10,10)`.

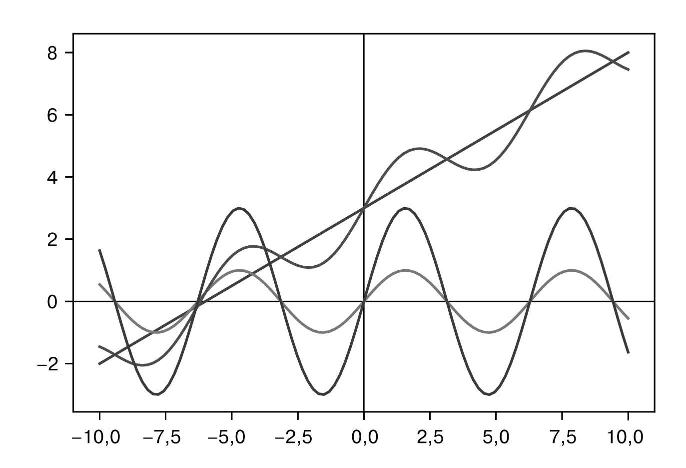
Решение

```
class Function(Vector):
    def __init__(self, f):
        self.function = f
    def add(self, other):
        return Function(lambda x: self.function(x) + other.function(x))
    def scale(self, scalar):
        return Function(lambda x: scalar * self.function(x))
    @classmethod
    def zero(cls):
        return Function(lambda x: 0)
    def __call__(self, arg):
        return self.function(arg)

f = Function(lambda x: 0.5 * x + 3)
g = Function(sin)

plot([f, g, f+g, 3*g], -10, 10)
```

Результат выполнения последней строки кода показан на графике.



Объекты `f` и `g` ведут себя подобно векторам, поэтому их можно складывать и умножать на скаляр. Кроме того, они ведут себя как функции, поэтому есть возможность построить их графики.

Упражнение 6.11. Мини-проект. Проверить равенство функций сложно. Используйте все свои знания и умения, чтобы написать функцию, проверяющую равенство двух функций.

Решение. Поскольку мы чаще сталкиваемся с гладкими непрерывными функциями, достаточно проверить близость их значений для нескольких случайных значений аргументов, как показано здесь:

```
def approx_equal_function(f,g):
    results = []
    for _ in range(0,10):
        x = uniform(-10,10)
        results.append(isclose(f(x),g(x)))
    return all(results)
```

К сожалению, результаты такой проверки могут вводить в заблуждение. Например, следующий вызов вернет `True` даже при том, что первая из функций не может быть равна нулю:

```
approx_equal_function(lambda x: (x*x)/x, lambda x: x)
```

Оказывается, вычисление равенства функций — *неразрешимая* задача. То есть было доказано, что не существует алгоритма, гарантирующего равенство любых двух функций.

Упражнение 6.12. Мини-проект. Выполните модульное тестирование класса `Function` и покажите, что функции удовлетворяют свойствам векторного пространства.

Решение. Проверить равенство функций трудно, не менее трудно сгенерировать случайные функции. Здесь я использовал класс `Polynomial` (с которым вы познакомитесь в следующем разделе), чтобы сгенерировать несколько случайных полиномиальных функций. Применив функцию `proc_equal_function` из предыдущего мини-проекта, можно гарантировать успешное прохождение теста:

```
def random_function():
    degree = randint(0,5)
    p = Polynomial(*[uniform(-10,10) for _ in range(0,degree)])
    return Function(lambda x: p(x))

for i in range(0,100):
    a,b = random_scalar(), random_scalar()
    u,v,w = random_function(), random_function(), random_function()
    test(Function.zero(), approx_equal_function, a,b,u,v,w)
```

Упражнение 6.13. Мини-проект. Реализуйте класс `Function2(Vector)`, представляющий функции двух переменных, например $f(x, y) = x + y$.

Решение. Определение этого класса не сильно отличается от определения класса `Function`, просто все функции получают по два аргумента:

```
class Function2(Vector):
    def __init__(self, f):
        self.function = f
    def add(self, other):
        return Function(lambda x,y: self.function(x,y) + other
          .function(x,y))
    def scale(self, scalar):
        return Function(lambda x,y: scalar * self.function(x,y))
    @classmethod
    def zero(cls):
        return Function(lambda x,y: 0)
    def __call__(self, *args):
        return self.function(*args)
```

Например, сумма $f(x, y) = x + y$ и $g(x, y) = x - y + 1$ должна дать в результате $2x + 1$. Вот как можно подтвердить это:

```
>>> f = Function2(lambda x,y:x+y)
>>> g = Function2(lambda x,y: x-y+1)
>>> (f+g)(3,10)
7
```

Упражнение 6.14. Какую размерность имеет векторное пространство матриц 9×9?

1. 9.
2. 18.
3. 27.
4. 81.

Решение. Матрица 9×9 содержит 81 элемент, поэтому она определяется 81 независимым числом (или координатой). Следовательно, это 81-мерное векторное пространство и правильный ответ — 4.

Упражнение 6.15. Мини-проект. Реализуйте класс `Matrix`, наследующий `Vector`, с абстрактными свойствами, представляющими количество строк и столбцов. Класс `Matrix` не позволяет создавать экземпляры, но можно создать, например, класс `Matrix5_by_3`, унаследовав `Matrix` и явно указав количество строк и столбцов.

Решение

```
class Matrix(Vector):
    @abstractproperty
    def rows(self):
        pass
    @abstractproperty
    def columns(self):
        pass
    def __init__(self,entries):
        self.entries = entries
    def add(self,other):
        return self.__class__(
            tuple(
                tuple(self.entries[i][j] + other.entries[i][j]
                      for j in range(0,self.columns()))
                for i in range(0,self.rows())))
    def scale(self,scalar):
        return self.__class__(
            tuple(
                tuple(scalar * e for e in row)
                for row in self.entries))
    def __repr__(self):
        return "%s%r" % (self.__class__.__qualname__, self.entries)
    def zero(self):
        return self.__class__(
            tuple(
                tuple(0 for i in range(0,self.columns()))
                for j in range(0,self.rows())))
```

С этим классом можно быстро реализовать любой класс, представляющий векторное пространство матриц фиксированного размера, например 2×2:

```
class Matrix2_by_2(Matrix):
    def rows(self):
        return 2
    def columns(self):
        return 2
```

и выполнять вычисления с матрицами 2×2 как с векторами:

```
>>> 2 * Matrix2_by_2(((1,2),(3,4))) + Matrix2_by_2(((1,2),(3,4)))
Matrix2_by_2((3, 6), (9, 12))
```

Упражнение 6.16. Выполните модульное тестирование класса `Matrix5_by_3`, чтобы продемонстрировать, что он подчиняется определяющим свойствам векторных пространств.

Решение

```
def random_matrix(rows, columns):
    return tuple(
        tuple(uniform(-10,10) for j in range(0,columns))
        for i in range(0,rows)
    )

def random_5_by_3():
    return Matrix5_by_3(random_matrix(5,3))

def approx_equal_matrix_5_by_3(m1,m2):
    return all([
        isclose(m1.matrix[i][j],m2.matrix[i][j])
        for j in range(0,3)
        for i in range(0,5)
    ])

for i in range(0,100):
    a,b = random_scalar(), random_scalar()
    u,v,w = random_5_by_3(), random_5_by_3(), random_5_by_3()
    test(Matrix5_by_3.zero(), approx_equal_matrix_5_by_3, a,b,u,v,w)
```

Упражнение 6.17. Мини-проект. Напишите класс `LinearMap3d_to_5d`, наследующий `Vector`, который хранит матрицу 5×3 и реализует метод `__call__`, выполняющий линейное преобразование $\mathbb{R}^3$ в $\mathbb{R}^5$. Покажите, что он согласуется с `Matrix5_by_3` в базовых вычислениях и проходит проверку определяющих свойств векторного пространства.

Упражнение 6.18. Мини-проект. Напишите функцию на Python, умножающую объекты `Matrix5_by_3` на объекты `Vec3` в соответствии с правилами умножения матриц. Дополните реализацию перегруженного оператора * для классов векторов и матриц, чтобы он позволял умножать векторы на матрицы или на скаляры слева от них.

Упражнение 6.19. Убедитесь, что сложение любого изображения `ImageVector` с нулевым вектором не создает никаких видимых изменений.

Решение. Возьмите любое изображение по своему выбору и выведите результат `ImageVector("my_image.jpg") + ImageVector.zero()`.

Упражнение 6.20. Выберите два изображения и покажите 10 разных средневзвешенных их значений. Это будут точки на отрезке, соединяющем изображения в 270 000-мерном пространстве!

Решение. Я запустил следующий код с s = 0,1, 0,2, 0,3, ..., 0,9, 1,0:

```
s * ImageVector("inside.JPG") + (1-s) * ImageVector("outside.JPG")
```

Поместив полученные изображения друг за другом, вы получите нечто похожее на следующий рисунок.

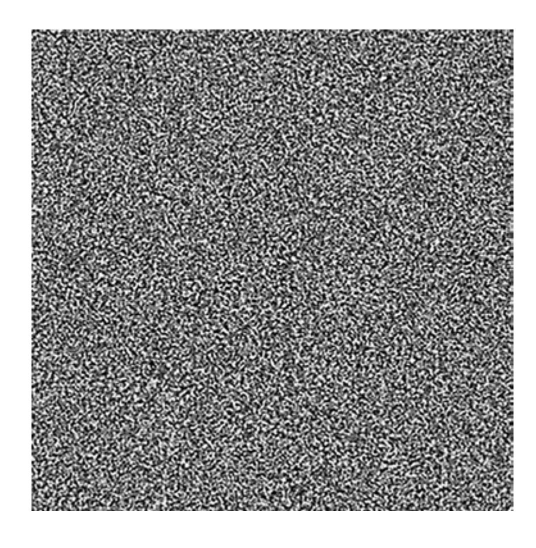


Несколько разных средневзвешенных значений двух изображений

Упражнение 6.21. Адаптируйте модульные тесты для проверки свойств векторного пространства изображений и выполните их. Как выглядят ваши случайные изображения, генерируемые модульными тестами?

Решение. Один из способов сгенерировать случайные изображения — присвоить каждому пикселу случайные значения красной, зеленой и синей составляющих, например:

```
def random_image():
    return ImageVector([(randint(0,255), randint(0,255), randint(0,255))
                        for i in range(0,300 * 300)])
```



В результате получатся беспорядочно распределенные цветные пикселы, но это не имеет никакого значения. Модульные тесты сравнивают каждый пиксел. Для тестирования можно использовать следующую функцию проверки на равенство:

```
def approx_equal_image(i1,i2):
    return all([isclose(c1,c2)
                for p1,p2 in zip(i1.pixels,i2.pixels)
                for c1,c2 in zip(p1,p2)])

for i in range(0,100):
    a,b = random_scalar(), random_scalar()
    u,v,w = random_image(), random_image(), random_image()
    test(ImageVector.zero(), approx_equal_image, a,b,u,v,w)
```

6.3. ПОИСК МЕНЬШИХ ВЕКТОРНЫХ ПРОСТРАНСТВ

Векторное пространство цветных изображений 300 × 300 имеет колоссальные 270 000 измерений — именно столько чисел необходимо, чтобы определить любое изображение такого размера. Объем данных сам по себе не проблема, но когда имеются большие изображения или большое количество изображений либо требуется объединить тысячи изображений в фильм, то суммарный объем данных может оказаться весьма внушительным.

В этом разделе мы посмотрим, как, начав с некоторого векторного пространства, найти меньшее векторное пространство (с меньшим числом измерений), сохраняющее бóльшую часть информации, имеющейся в исходном пространстве. В случае с изображениями можно уменьшить количество отдельных пикселов в изображении или преобразовать цветное изображение в черно-белое. Результат может получиться не идеальным, но все же узнаваемым. Например, изображение, что на рис. 6.12, *справа*, состоит из 900 чисел и получено из фото, размещенного слева и состоящего из 270 000 чисел.

Изображение справа находится в 900-мерном *подпространстве* 270 000-мерного пространства. То есть оно по-прежнему является 270 000-мерным вектором, но может быть представлено или сохранено всего лишь 900 координатами. Это отправная точка для изучения *сжатия*. Мы не будем слишком углубляться в практические приемы сжатия, а лишь познакомимся с подпространствами векторных пространств.




Рис. 6.12. Результат преобразования изображения, заданного 270 000 числами *(слева)*, в изображение, заданное 900 числами *(справа)*

6.3.1. Идентификация подпространств

Векторное *подпространство*, или просто подпространство, — это именно то, что подразумевается: векторное пространство, существующее внутри другого векторного пространства. Один из примеров, который мы уже рассматривали много раз, — двухмерная плоскость xy в трехмерном пространстве, такая как плоскость $z = 0$. Точнее, подпространство состоит из векторов формы $(x, y, 0)$. Эти векторы имеют три координаты и поэтому являются настоящими трехмерными векторами, но образуют подмножество векторов, лежащих на плоскости. По этой причине мы говорим, что это двухмерное подпространство $\mathbb{R}^3$.

ПРИМЕЧАНИЕ

> С технической точки зрения двухмерное векторное пространство $\mathbb{R}^2$, состоящее из упорядоченных пар (x, y), не является подпространством трехмерного пространства $\mathbb{R}^3$, потому что векторы с формой (x, y) не являются трехмерными векторами. Однако оно имеет взаимно однозначное соответствие с набором векторов $(x, y, 0)$, и векторная арифметика выглядит одинаковой независимо от наличия лишней нулевой координаты z. По этим причинам я считаю правильным называть $\mathbb{R}^2$ подпространством $\mathbb{R}^3$.

Не каждое подмножество трехмерных векторов является подпространством. Плоскость $z = 0$ — особенная, потому что векторы $(x, y, 0)$ образуют автономное векторное пространство. Невозможно построить линейную комбинацию векторов в этой плоскости, которая каким-то образом дает вектор, находящийся за ее пределами, — третья координата всегда остается нулевой. На математическом языке такие самодостаточные подпространства называют *закрытыми* в отношении линейных комбинаций.

Чтобы понять, как вообще выглядит векторное подпространство, найдем подмножества векторных пространств, которые также являются подпространствами

(рис. 6.13). Какие подмножества векторов на плоскости могут составить автономное векторное пространство? Можно ли просто нарисовать любую область на плоскости и взять только те векторы, которые находятся внутри нее?

Ответ: нет. Подмножество на рис. 6.13 содержит несколько векторов, лежащих на оси x, и несколько векторов, лежащих на оси y. Их можно масштабировать, чтобы получить векторы стандартного базиса $\mathbf{e}_1 = (1, 0)$ и $\mathbf{e}_2 = (0, 1)$. Из этих векторов можно составить линейные комбинации, позволяющие попасть в любую точку плоскости, а не только в S (рис. 6.14).

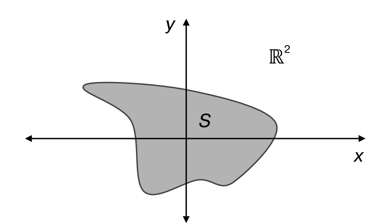


Рис. 6.13. *S* — подмножество точек (векторов) на плоскости $\mathbb{R}^2$. Является ли *S* подпространством $\mathbb{R}^2$?

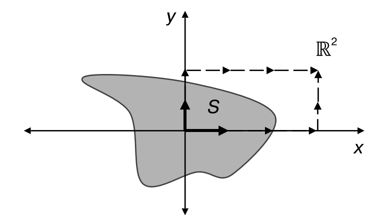


Рис. 6.14. Линейная комбинация двух векторов в *S* дает «путь выхода» за пределы *S*. То есть *S* не может быть подпространством

Вместо рисования случайных подпространств воспроизведем пример плоскости в трехмерном пространстве. Координаты z у нас нет, поэтому выберем все точки с координатой $y = 0$. В результате получим множество точек на оси x, имеющее форму $(x, 0)$. Как бы мы ни старались, мы не можем найти линейную комбинацию векторов этой формы, дающую вектор с ненулевой координатой y (рис. 6.15).

Эта прямая, $y = 0$, является векторным подпространством пространства $\mathbb{R}^2$. Первоначально мы нашли двухмерное подпространство в трехмерном пространстве, затем точно так же нашли одномерное подпространство в двухмерном пространстве. В отличие от трехмерного *пространства* или двухмерной *плоскости*, одномерное векторное пространство называется *прямой линией*. Фактически мы можем идентифицировать это подпространство как числовую прямую $\mathbb{R}$.

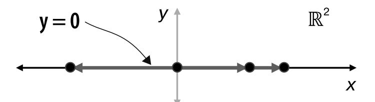


Рис. 6.15. Сосредоточившись на множестве точек с $y = 0$, получаем векторное пространство, содержащее все линейные комбинации своих точек

Следующий шаг — фиксация координаты $x = 0$. После фиксации координат $x = 0$ и $y = 0$ у нас остается только одна точка — нулевой вектор. Это тоже векторное подпространство! Любые линейные комбинации с нулевым вектором будут давать в результате нулевой вектор. Это *нульмерное подпространство* на

одномерной прямой, в двухмерной плоскости и трехмерном пространстве. Геометрически нульмерное подпространство — это точка, и она должна быть нулем. Если бы это была какая-то другая точка, например v, она также содержала бы $0 \cdot v = 0$ и бесконечное количество других скалярных множителей, таких как $3 \cdot v$ и $-42 \cdot v$. Исследуем эту идею.

6.3.2. Начнем с единственного вектора

Векторное подпространство, содержащее ненулевой вектор **v**, включает также все произведения **v** на скаляр. Геометрически множество всех скалярных множителей ненулевого вектора **v** лежит на прямой, проходящей через начало координат, как показано на рис. 6.16.

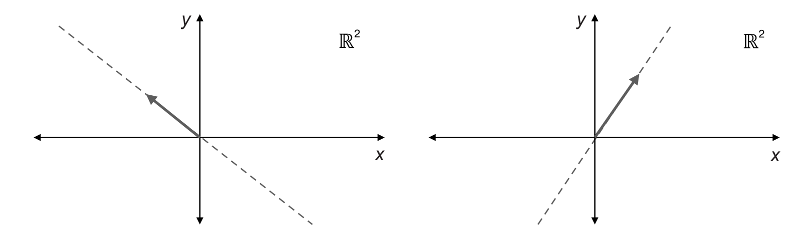


Рис. 6.16. Два разных вектора на пунктирных прямых, показывающих, где будут располагаться все их скалярные множители

Каждая из этих прямых, проходящих через начало координат, — векторное пространство. Невозможно получить вектор за пределами такой прямой, используя лишь сложение и умножение на скаляр векторов на этой прямой. Это верно и для линий, проходящих через начало координат в трехмерном пространстве: все они являются линейными комбинациями одного трехмерного вектора и образуют векторное пространство. Это первый пример универсального способа построения подпространств — выбор вектора и просмотр всех линейных комбинаций с ним.

6.3.3. Охват большего пространства

Пространство, *охватываемое* заданным множеством из одного или нескольких векторов (span, еще называется *линейной оболочкой*), определяется как множество всех линейных комбинаций. Одна из важнейших особенностей этого пространства — оно автоматически становится векторным подпространством. Иначе говоря, пространство, охватываемое одним вектором **v**, — это прямая, проходящая через начало координат. Мы обозначаем множество объектов, заключая их в фигурные скобки, поэтому множество, содержащее только **v**, можно записать как $\{\mathbf{v}\}$, а пространство, охватываемое этим множеством, — как $\text{span}(\{\mathbf{v}\})$.

После включения в множество другого вектора **w**, не параллельного вектору **v**, пространство становится больше, потому что теперь мы не ограничены одним линейным направлением. Пространство множества двух векторов {**v**, **w**} включает две прямые, span({**v**}) и span({**w**}), а также линейные комбинации, включающие **v** и **w**, которые не лежат ни на одной из заданных прямых (рис. 6.17).

Это может быть неочевидно, но пространством, охватываемым этими двумя векторами, является вся плоскость. То же верно для любой пары непараллельных векторов на плоскости, но особенно ярко заметно на примере векторов стандартного базиса. Любую точку (x, y) можно получить как линейную комбинацию $x \cdot (1, 0) + y \cdot (0, 1)$. То же верно и для других пар непараллельных векторов, таких как **v** = (1, 0) и **w** = (1, 1), но, чтобы увидеть это, нужно приложить чуть больше усилий.

Мы можем получить любую точку, например (4, 3), использовав правильную линейную комбинацию векторов (1, 0) и (1, 1). Единственный способ получить координату y, равную 3, — трижды отложить вектор (1, 1). В результате получится (3, 3). Чтобы из этой точки попасть в точку (4, 3), нужно отложить вектор (1, 0). Таким образом мы получаем линейную комбинацию $3 \cdot (1, 1) + 1 \cdot (1, 0)$, которая ведет к точке (4, 3), как показано на рис. 6.18.

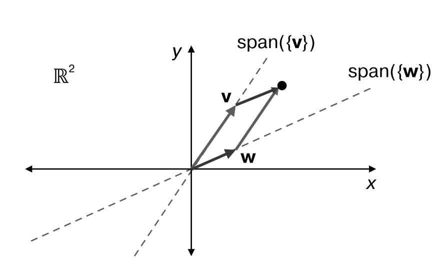


Рис. 6.17. Пространство, охватываемое двумя непараллельными векторами. Каждый отдельный вектор охватывает прямую, но вместе они охватывают более обширное множество точек: например, **v** + **w** не лежит ни на одной из прямых

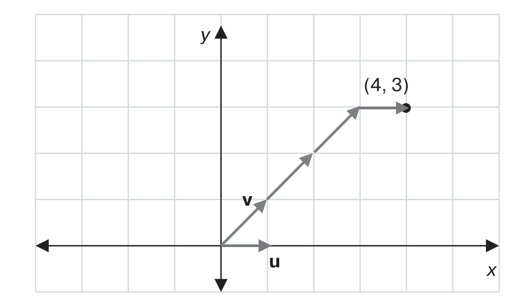


Рис. 6.18. Как добраться до произвольной точки (4, 3) с помощью линейной комбинации (1, 0) и (1, 1)

Один ненулевой вектор охватывает прямую в двух- или трехмерном пространстве, и оказывается, что два непараллельных вектора могут охватывать либо всю двухмерную плоскость, либо плоскость, проходящую через начало координат в трехмерном пространстве. На рис. 6.19 показано, как может выглядеть плоскость, натянутая на два трехмерных вектора.

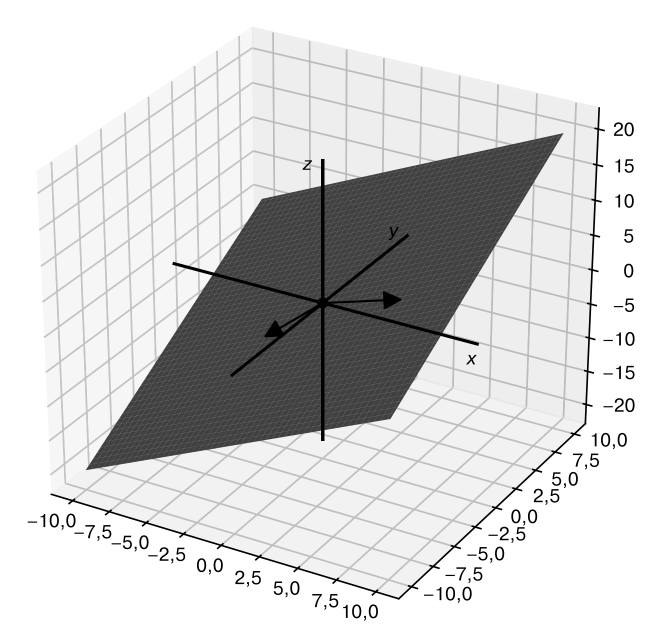


Рис. 6.19. Плоскость, натянутая на два трехмерных вектора

Она наклонена, потому что отличается от плоскости $z = 0$ и не содержит ни одного из трех векторов стандартного базиса. Тем не менее это плоскость и векторное подпространство в трехмерном пространстве. Один вектор образует одномерное пространство, а два непараллельных вектора — двухмерное пространство. Если добавить в это множество третий непараллельный вектор, то образует ли оно трехмерное пространство? На рис. 6.20 ясно видно, что нет.

Никакие два вектора из тройки **u**, **v** и **w** не параллельны друг другу, но они не образуют трехмерного пространства. Все они располагаются на двухмерной плоскости, поэтому никакая их линейная комбинация не может волшебным образом дать координату *z*. Нам нужно какое-то другое, более удачное обобщение понятия непараллельных векторов.

Если требуется добавить в множество вектор и охватить пространство более высокой размерности, новый вектор должен указывать в новом направлении, за пределы пространства, охватываемого существующими векторами. Три вектора на плоскости несколько избыточны. Например, как показано на рис. 6.21, линейная комбинация **u** и **w** дает **v**.

Правильное обобщение непараллельности — *линейная независимость*. Набор векторов считается *линейно зависимым*, если любой из его элементов можно получить как линейную комбинацию других элементов. Два параллельных

вектора линейно зависимы, потому что они скалярно кратны друг другу. Точно так же множество из трех векторов {**u**, **v**, **w**} линейно зависимо, потому что вектор **v** можно получить из линейной комбинации **u** и **w** (или **w** из линейной комбинации **u** и **v** и т. д.). Вы должны прочувствовать эту концепцию. В одном из упражнений в конце этого раздела вам будет предложено проверить, что любой из трех векторов, (1, 0), (1, 1) и (−1, 1), можно записать как линейную комбинацию двух других.

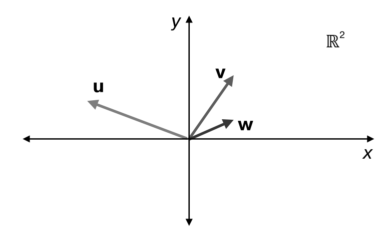


Рис. 6.20. Три непараллельных вектора, образующих только двухмерное пространство

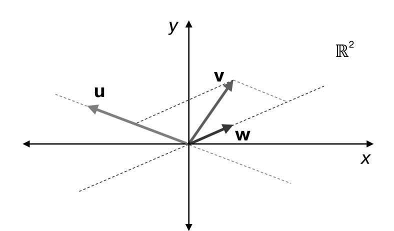


Рис. 6.21. Линейная комбинация **u** и **w** дает **v**, поэтому пространство, образованное тройкой векторов **u**, **v** и **w**, не может быть больше пространства, охватываемого парой **u** и **w**

Множество {**u**, **v**}, напротив, *линейно независимо*, потому что его элементы непараллельны и ни один из них нельзя выразить через простое умножение другого на скаляр. Это означает, что **u** и **v** охватывают большее пространство, чем каждый из них сам по себе. Точно так же стандартный базис $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$ для $\mathbb{R}^3$ — это линейно независимое множество. Ни один из этих векторов нельзя получить как линейную комбинацию двух других, и все три необходимы для охвата трехмерного пространства. Мы подходим к свойствам векторного пространства или подпространства, которые указывают на его размерность.

6.3.4. Определение размерности

Попробуйте ответить на вопрос: является ли множество трехмерных векторов

$$\{(1, 1, 1), (2, 0, -3), (0, 0, 1), (-1, -2, 0)\}$$

линейно независимым? Чтобы сделать это, можно нарисовать векторы в трехмерном пространстве или попытаться найти линейную комбинацию трех из них, чтобы получить четвертый. Но есть более простой ответ: для охвата всего трехмерного пространства необходимо и достаточно трех векторов, поэтому любое множество из четырех трехмерных векторов будет несколько избыточным.

Мы знаем, что множество с одним или двумя трехмерными векторами охватывает прямую или плоскость соответственно, а не все пространство $\mathbb{R}^3$. Три — это магическое число векторов, которые могут охватывать трехмерное пространство и при этом оставаться линейно независимыми. Именно поэтому мы называем его трехмерным: в нем три независимых направления.

Линейно независимое множество векторов, охватывающее все векторное пространство, например $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$ для $\mathbb{R}^3$, называется *базисом*. Любой базис пространства имеет одинаковое количество векторов, и это число — его *размерность*. Мы видели, что (1, 0) и (1, 1) линейно независимы и охватывают всю плоскость, поэтому они являются основой для векторного пространства $\mathbb{R}^2$. Точно так же (1, 0, 0) и (0, 1, 0) линейно независимы и охватывают плоскость $z = 0$ в $\mathbb{R}^3$, что делает их основой для этого двухмерного подпространства, но не для всего $\mathbb{R}^3$.

Я уже использовал слово *«базис»* в контексте «стандартный базис» для $\mathbb{R}^2$ и $\mathbb{R}^3$. Он называется стандартным, потому что является естественным выбором. Разложение вектора координат в стандартном базисе не требует вычислений — координаты *являются* скалярами в этом разложении. Например, (3, 2) означает линейную комбинацию $3 \cdot (1, 0) + 2 \cdot (0, 1)$, или $3\mathbf{e}_1 + 2\mathbf{e}_2$.

В общем случае, чтобы определить, являются ли векторы линейно независимыми, требуется выполнить некоторые вычисления. Даже если известно, что вектор — линейная комбинация некоторых других векторов, для ее нахождения нужно произвести некоторые алгебраические вычисления. В следующей главе я расскажу, как это сделать, — это распространенная вычислительная задача линейной алгебры. Но прежде еще немного попрактикуемся в определении подпространств и их размерностей.

6.3.5. Определение подпространств векторного пространства функций

Математические функции из $\mathbb{R}$ в $\mathbb{R}$ содержат бесконечное количество данных — выходных значений, образующихся при получении любого из бесконечного множества действительных чисел на входе. Однако это не означает, что для описания функций необходимо бесконечное количество данных. Например, чтобы описать линейную функцию, достаточно двух действительных чисел — значений a и b в широко известной формуле

$$f(x) = ax + b,$$

где a и b могут быть любыми действительными числами. Это гораздо удобнее бесконечномерного пространства всех функций. Любую линейную функцию можно задать двумя действительными числами, поэтому, по всей видимости, подпространство линейных функций двухмерно.

ВНИМАНИЕ

В последних главах я использовал слово «линейный» во многих новых контекстах. Здесь я возвращаюсь к тому значению, которое нам дали в школе: линейная функция — это функция, график которой имеет форму прямой линии. К сожалению, функции этого вида не являются линейными в том смысле, который подразумевался на протяжении всей главы 4, и вы можете доказать это, выполнив предложенное упражнение. По этой причине я постараюсь четко указывать, в каком смысле употребляется слово «линейный» в тот или иной момент.

Мы можем быстро реализовать класс `LinearFunction`, наследующий `Vector`, и вместо самой функции хранить два числа — коэффициенты a и b. Такие функции можно складывать, складывая их коэффициенты, потому что:

$$(ax + b) + (cx + d) = (ax + cx) + (b + d) = (a + c)x + (b + d),$$

и умножать на скаляр, умножая на скаляр их коэффициенты: $r(ax + b) = rax + rb$. Наконец, оказывается, что нулевая функция $f(x) = 0$ тоже линейная. Это случай, когда $a = b = 0$. Вот ее реализация:

```
class LinearFunction(Vector):
    def __init__(self,a,b):
        self.a = a
        self.b = b
    def add(self,v):
        return LinearFunction(self.a + v.a, self.b + v.b)
    def scale(self,scalar):
        return LinearFunction(scalar * self.a, scalar * self.b)
    def __call__(self,x):
        return self.a * x + self.b
    @classmethod
    def zero(cls):
        return LinearFunction(0,0)
```

Как показано на рис. 6.22, вызов `plot([LinearFunction(-2, 2)], -5, 5)` дает график прямой линии $f(x) = -2x + 2$.

Можно доказать, что линейные функции образуют векторное подпространство с размерностью 2, написав базис. Оба базисных вектора должны быть функциями, охватывать все пространство линейных функций и быть линейно независимыми (не кратными друг другу). Таким множеством является $\{x, 1\}$, или, более конкретно, $\{f(x) = x, g(x) = 1\}$. Соответственно, функцию вида $ax + b$ можно записать в виде линейной комбинации $a \cdot f + b \cdot g$.

Это очень близко к стандартному базису линейных функций: $f(x) = x$ и $g(x) = 1$ явно разные и не кратны друг другу, то есть ни одну из них нельзя выразить через другую умножением на скаляр. Напротив, функции $f(x) = x$ и $h(x) = 4x$ кратны

друг другу и не являются линейно независимой парой. Но $\{x, 1\}$ — не единственный базис, который мы могли бы выбрать, $\{4x + 1, x - 3\}$ тоже может играть роль базиса.

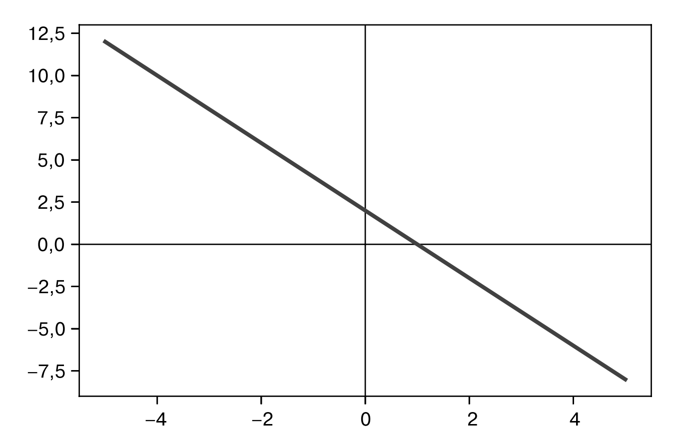


Рис. 6.22. График линейной функции LinearFunction(-2, 2), представляющей $f(x) = -2x + 2$

То же самое относится к квадратичным функциям, имеющим вид $f(x) = ax^2 + bx + c$. Они образуют трехмерное подпространство векторного пространства функций с одним из вариантов базиса $\{x^2, x, 1\}$. Линейные функции образуют векторное подпространство пространства квадратичных функций, где компонента x^2 равна нулю. Линейные и квадратичные функции являются примерами *полиномиальных функций* — линейных комбинаций степеней x, например:

$$f(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n.$$

Линейные и квадратичные функции имеют *степень* 1 и 2 соответственно, потому что в каждой из них присутствуют высшие степени x. Полином в приведенном ранее уравнении имеет степень n и $n + 1$ коэффициентов. В упражнениях вы увидите, что пространство полиномов *любой* степени образует свое векторное подпространство в пространстве функций.

6.3.6. Подпространства изображений

Наши объекты `ImageVector` представляют собой 270 000 чисел, поэтому мы могли бы использовать стандартную формулу и построить базис из 270 000 изображений, в каждом из которых одно из 270 000 чисел равно единице, а все остальные — нулю. В листинге 6.4 показано, как мог бы выглядеть первый базисный вектор.

Листинг 6.4. Псевдокод, конструирующий первый базисный вектор

```
ImageVector([
    (1,0,0), (0,0,0), (0,0,0), ..., (0,0,0),
    (0,0,0), (0,0,0), (0,0,0), ..., (0,0,0),
    ...
])
```

Только первый пиксел в первой строке не равен нулю — он имеет значение 1 в красной составляющей цвета. Все остальные пикселы имеют значение (0,0,0)

Вторая строка состоит из 300 черных пикселов, каждый из которых имеет значение (0,0,0)

Я опустил остальные 298 строк: они идентичны второй строке — все пикселы черные

Этот единственный вектор охватывает одномерное подпространство, состоящее из изображений черного цвета с одним красным пикселом в верхнем левом углу. Умножение этого вектора на скаляр будет менять яркость красного пиксела, но все остальные пикселы так и останутся черными. Чтобы показать больше пикселов, нужно больше базисных векторов.

Запись этих 270 000 базисных векторов не даст нам ничего нового. Поэтому поищем меньшее множество векторов, охватывающих интересное для нас подпространство. Вот, например, экземпляр `ImageVector`, состоящий из темно-серых пикселов:

```
gray = ImageVector([
    (1,1,1), (1,1,1), (1,1,1), ..., (1,1,1),
    (1,1,1), (1,1,1), (1,1,1), ..., (1,1,1),
    ...
])
```

Мы могли бы записать его более кратко:

```
gray = ImageVector([(1,1,1) for _ in range(0,300*300)])
```

Один из способов увидеть подпространство, образуемое единственным вектором `gray`, — изобразить некоторые принадлежащие ему векторы. На рис. 6.23 показаны произведения вектора `gray` на скаляры.

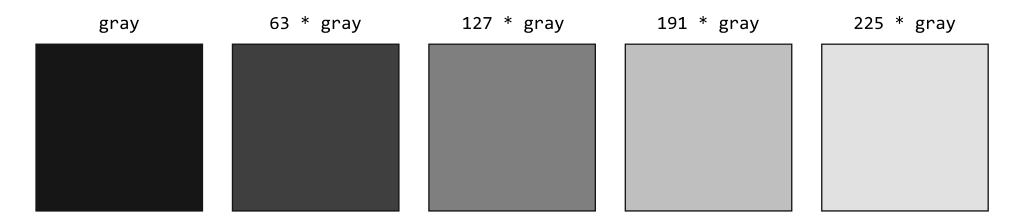


Рис. 6.23. Некоторые из векторов в одномерном подпространстве, образованном экземпляром gray класса ImageVector

Это одномерная коллекция изображений, если говорить простым языком. Здесь меняется только одна характеристика — яркость.

На это подпространство можно взглянуть также с точки зрения значений пикселов: все пикселы в любом изображении имеют одинаковые значения. Для любого конкретного пиксела существует трехмерное пространство цветовых возможностей, измеряемое координатами красного, зеленого и синего цветов. Серые пикселы образуют одномерное подпространство этого пространства, содержащее точки со всеми координатами $s \cdot (1, 1, 1)$ для некоторых скаляров s (рис. 6.24).

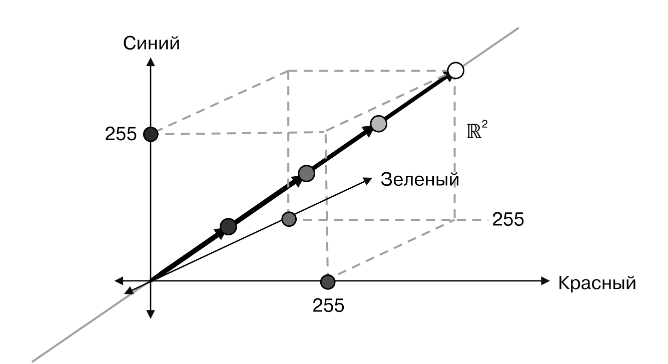


Рис. 6.24. Серые пикселы разной яркости на одной прямой. Серые пикселы образуют одномерное подпространство в трехмерном векторном пространстве значений пикселов

Все изображения в базисе будут черными, за исключением одного пиксела — очень тусклого красного, зеленого или синего. Изменение одного пиксела за раз не дает впечатляющих результатов, поэтому поищем меньшие и более интересные подпространства.

Существует множество подпространств изображений, которые можно было бы исследовать. Например, можно рассмотреть сплошные цветные изображения любого цвета:

```
ImageVector([
    (r,g,b), (r,g,b), (r,g,b), ..., (r,g,b),
    (r,g,b), (r,g,b), (r,g,b), ..., (r,g,b),
    ...
])
```

На конкретные значения пикселов нет никаких ограничений, единственное ограничение, которое следует соблюдать для получения сплошного цветного

изображения, — все пикселы должны иметь одинаковые значения. В качестве последнего примера рассмотрим подпространство черно-белых изображений с низким разрешением (рис. 6.25).

Все пикселы в каждом блоке размером 10×10 имеют одно и то же значение, что делает изображение похожим на сетку 30×30. Всего имеется $30 \cdot 30 = 900$ чисел, определяющих это изображение, поэтому подобные изображения образуют 900-мерное подпространство 270 000-мерного пространства изображений. Это подпространство содержит намного меньше данных, но все еще способно создавать узнаваемые изображения.

Один из способов получить изображение в этом подпространстве — взять за основу любое изображение с более высоким разрешением и усреднить все значения красного, зеленого и синего в каждом блоке 10×10 пикселов. Это среднее даст яркость b, и вы сможете задать для всех пикселов в блоке значение (b, b, b), чтобы получить новое изображение. Это линейное отображение (рис. 6.26), и позже вам будет дана возможность реализовать его в виде мини-проекта.

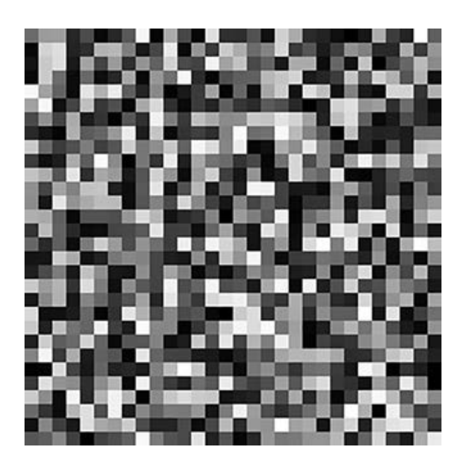


Рис. 6.25. Черно-белое изображение низкого разрешения. Пикселы в каждом блоке 10×10 имеют одинаковые значения

Моя собака Мельба выглядит на втором снимке не так фотогенично, как на первом, но все же вполне узнаваема. Это пример, о котором я упоминал в начале раздела, но самое замечательное то, что мы все еще можем сказать: это то же самое изображение, хотя в нем осталось всего 0,3 % от исходных данных. Очевидно, что это не самое совершенное решение, и все же подход к отображению в подпространство может послужить отправной точкой для более плодотворного исследования. В главе 13 я покажу, как таким образом сжимать аудиоданные..

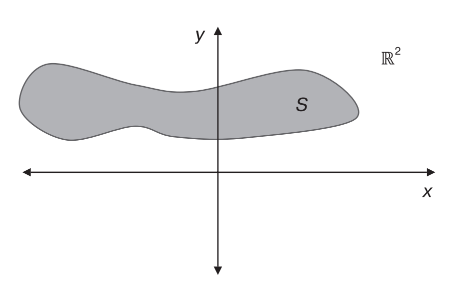




Рис. 6.26. Линейное отображение превращает любое существующее изображение *(слева)* в новое изображение *(справа)*, лежащее в 900-мерном подпространстве

6.3.7. Упражнения

Упражнение 6.22. Докажите геометрически, почему следующая область S плоскости не может считаться векторным подпространством этой плоскости.



Решение. В этой области есть множество точек, линейные комбинации которых выходят за границы области. Еще один очевидный факт, доказывающий, что эта область не может быть векторным пространством, — она не включает нулевой вектор. Нулевой вектор — это результат умножения любого вектора на скалярный нуль, поэтому он должен входить в любое векторное пространство или подпространство.

Упражнение 6.23. Покажите, что область плоскости с $x = 0$ образует одномерное векторное пространство.

Решение. Эту область составляют векторы, лежащие на оси y и имеющие форму $(0, y)$ для любого действительного числа y. Сложение и умножение на скаляр векторов вида $(0, y)$ выполняется так же, как и для действительных чисел, просто в формуле случайно оказался лишний 0. Мы можем заключить, что это завуалированное пространство $\mathbb{R}$ и, следовательно, одномерное векторное пространство. Для большей строгости можно явно проверить все свойства векторного пространства.

Упражнение 6.24. Покажите, что три вектора, $(1, 0)$, $(1, 1)$ и $(-1, 1)$, линейно зависимы, записав каждый из них как линейную комбинацию двух других.

Решение

$$(1, 0) = 1/2 \cdot (1, 1) - 1/2 \cdot (-1, 1);$$

$$(1, 1) = 2 \cdot (1, 0) + (-1, 1);$$

$$(-1, 1) = (1, 1) - 2 \cdot (1, 0).$$

Упражнение 6.25. Покажите, что любой вектор (x, y) можно получить как линейную комбинацию векторов (1, 0) и (1, 1).

Решение. Мы знаем, что (1, 0) не может вносить вклад в координату y, поэтому нужно y раз отложить вектор (1, 1). А чтобы достичь требуемой точки, следует $(x - y)$ раз отложить вектор (1, 0):

$$(x, y) = (x - y) \cdot (1, 0) + y \cdot (1, 1).$$

Упражнение 6.26. Объясните на примере, почему для одного вектора **v** множество всех линейных комбинаций **v** совпадает со множеством всех произведений **v** на скаляр.

Решение. Линейные комбинации вектора и он сам сводятся к произведению на скаляр в соответствии с одним из признаков векторного пространства. Например, линейная комбинация $a \cdot \mathbf{v} + b \cdot \mathbf{v}$ равна $(a + b) \cdot \mathbf{v}$.

Упражнение 6.27. Объясните с геометрической точки зрения, почему прямая, не проходящая через начало координат, не является векторным подпространством плоскости или трехмерного пространства.

Решение. Такая прямая не может быть подпространством по одной простой причине — она проходит не через начало координат (нулевой вектор). Другая причина: такая прямая будет иметь два непараллельных вектора. А как мы знаем, непараллельные векторы охватывают всю плоскость, которая намного больше прямой.

Упражнение 6.28. Никакие два вектора из $\{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$ не охватывают все пространство $\mathbb{R}^3$ — они охватывают только двухмерные подпространства трехмерного пространства. Что это за подпространства?

Решение. Множество $\{\mathbf{e}_1, \mathbf{e}_2\}$ охватывает все линейные комбинации $a \cdot \mathbf{e}_1 + b \cdot \mathbf{e}_2$ или $a \cdot (1, 0, 0) + b \cdot (0, 1, 0) = (a, b, 0)$. В зависимости от выбора a и b это может быть любая точка на плоскости $z = 0$, часто называемой плоскостью xy. Аналогично, векторы $\{\mathbf{e}_2, \mathbf{e}_3\}$ охватывают плоскость $x = 0$, называемую плоскостью yz, а векторы $\{\mathbf{e}_1, \mathbf{e}_3\}$ — плоскость $y = 0$, называемую плоскостью xz.

Упражнение 6.29. Представьте вектор (–5, 4) как линейную комбинацию векторов (0, 3) и (–2, 1).

Решение. Только (–2, 1) может дать вклад в координату x, поэтому нам нужно, чтобы в сумме было 2,5 · (–2, 1). Это приводит нас к вектору (–5, 2,5), поэтому нужны дополнительные 1,5 единицы по координате x, или 0,5 · (0, 3). Искомая линейная комбинация будет иметь вид

$$(-5, 4) = 0{,}5 \cdot (0, 3) + 2{,}5 \cdot (-2, 1).$$

Упражнение 6.30. Мини-проект. Являются ли векторы (1, 2, 0), (5, 0, 5) и (2, –6, 5) линейно независимыми?

Решение. Решить эту задачу непросто, но все же существует линейная комбинация первых двух векторов, дающая третий:

$$-3 \cdot (1, 2, 0) + (5, 0, 5) = (2, -6, 5).$$

Это означает, что третий вектор избыточен, а векторы линейно зависимы. Они охватывают только двухмерное подпространство трехмерного пространства, а не все трехмерное пространство.

Упражнение 6.31. Объясните, почему линейная функция $f(x) = ax + b$ не является линейным отображением векторного пространства $\mathbb{R}$ на самого себя, если только не выполняется условие $b = 0$.

Решение. Обратимся непосредственно к определению: линейное отображение должно сохранять линейные комбинации. Однако, как видите, f не сохраняет линейные комбинации действительных чисел. Например, $f(1 + 1) = 2a + b$, а $f(1) + f(1) = (a + b) + (a + b) = 2a + 2b$. Это условие не будет выполняться при $b = 0$.

Можно дать и другое объяснение: как известно, линейные функции $\mathbb{R} \to \mathbb{R}$ должны быть представлены матрицами 1 × 1. Матричное умножение одномерного вектора-столбца $[x]$ на матрицу 1 × 1 $[a]$ дает $[ax]$. Это необычный случай умножения матриц, но реализация из главы 5 подтверждает этот результат. Если функция $\mathbb{R} \to \mathbb{R}$ линейная, она должна согласовываться с матричным умножением 1 × 1 и, следовательно, быть умножением на скаляр.

Упражнение 6.32. Переопределите класс `LinearFunction` так, чтобы он наследовал `Vec2` и реализовал метод `__call__`.

Решение. Данные в `Vec2` определены под именами x и y вместо a и b, прочая функциональность остается прежней. Остается только реализовать `__call__`:

```
class LinearFunction(Vec2):
    def __call__(self,input):
        return self.x * input + self.y
```

Упражнение 6.33. Докажите (алгебраически!), что линейные функции вида $f(x) = ax + b$ образуют векторное подпространство векторного пространства всех функций.

Решение. Чтобы доказать это, нужно показать, что линейная комбинация двух линейных функций — это другая линейная функция. Пусть $f(x) = ax + b$ и $g(x) = cx + d$, тогда $r \cdot f + s \cdot g$ даст в результате

$$r \cdot f + s \cdot g = r \cdot (ax + b) + s \cdot (cx + d) = rax + b + scx + d = (ra + sc) \cdot x + (b + d).$$

Поскольку $(ra + sc)$ и $(b + d)$ — скаляры, мы получаем функцию нужного вида. Отсюда можно заключить, что линейные функции замкнуты относительно линейных комбинаций и, следовательно, образуют подпространство.

Упражнение 6.34. Найдите базис для набора матриц 3×3. Определите размерность этого векторного пространства.

Решение. Базис состоит из девяти матриц 3×3:

$$\begin{pmatrix}1&0&0\\0&0&0\\0&0&0\end{pmatrix}\begin{pmatrix}0&1&0\\0&0&0\\0&0&0\end{pmatrix}\begin{pmatrix}0&0&1\\0&0&0\\0&0&0\end{pmatrix}$$
$$\begin{pmatrix}0&0&0\\1&0&0\\0&0&0\end{pmatrix}\begin{pmatrix}0&0&0\\0&1&0\\0&0&0\end{pmatrix}\begin{pmatrix}0&0&0\\0&0&1\\0&0&0\end{pmatrix}.$$
$$\begin{pmatrix}0&0&0\\0&0&0\\1&0&0\end{pmatrix}\begin{pmatrix}0&0&0\\0&0&0\\0&1&0\end{pmatrix}\begin{pmatrix}0&0&0\\0&0&0\\0&0&1\end{pmatrix}$$

Они линейно независимы, каждая вносит свой вклад в любую линейную комбинацию. Они также охватывают целое пространство, потому что любая матрица может быть построена как их линейная комбинация, коэффициент в каждой конкретной матрице определяет один элемент результата. Поскольку эти девять векторов составляют базис пространства матриц 3×3, данное пространство имеет девять измерений.

Упражнение 6.35. Мини-проект. Реализуйте класс `QuadraticFunction(Vector)`, представляющий векторное подпространство функций вида $ax^2 + bx + c$. Что является базисом этого подпространства?

Решение. Реализация очень похожа на `LinearFunction`, только здесь три коэффициента, а не два, и функция `__call__` имеет квадратный член:

```
class QuadraticFunction(Vector):
    def __init__(self,a,b,c):
        self.a = a
        self.b = b
        self.c = c
    def add(self,v):
        return QuadraticFunction(self.a + v.a,
                                 self.b + v.b,
                                 self.c + v.c)
    def scale(self,scalar):
        return QuadraticFunction(scalar * self.a,
                                 scalar * self.b,
                                 scalar * self.c)
    def __call__(self,x):
        return self.a * x * x + self.b * x + self.c
    @classmethod
    def zero(cls):
        return QuadraticFunction(0,0,0)
```

Обратите внимание на то, что $ax^2 + bx + c$ выглядит как линейная комбинация множества $\{x^2, x, 1\}$. Эти три функции действительно охватывают пространство, и ни одну из них нельзя записать как линейную комбинацию остальных. Например, невозможно получить член x^2 сложением линейных функций. Следовательно, это базис. Поскольку векторов три, можно сделать вывод, что это трехмерное подпространство пространства функций.

Упражнение 6.36. Мини-проект. Ранее я заявлял, что набор $\{4x + 1, x - 2\}$ — это базис для множества линейных функций. Покажите, что можно записать $-2x + 5$ как линейную комбинацию этих двух функций.

Решение. $(1/9) \cdot (4x + 1) - (22/9) \cdot (x - 2) = -2x + 5$. Если вы еще не совсем забыли алгебру, то сможете решить эту задачу самостоятельно. Но если у вас что-то не получится, не волнуйтесь — в следующей главе я расскажу, как решать подобные задачи.

Упражнение 6.37. Мини-проект. Векторное пространство всех полиномов является бесконечномерным. Реализуйте это векторное пространство как класс и опишите базис (он должен быть бесконечным множеством!).

Решение

```
class Polynomial(Vector):
    def __init__(self, *coefficients):
        self.coefficients = coefficients
    def __call__(self,x):
        return sum(coefficient * x ** power
                   for (power,coefficient)
                   in enumerate(self.coefficients))
    def add(self,p):
        return Polynomial([a + b
                          for a,b
                          in zip(self.coefficients,
                                 p.coefficients)])
    def scale(self,scalar):
        return Polynomial([scalar * a
                          for a in self.coefficients])
    def _repr_latex_(self):
        monomials = [repr(coefficient) if power == 0
                                       else "x ^ {%d}" % power
                                           if coefficient == 1
                                       else "%s x ^ {%d}" % (coefficient,
                                           power)
                    for (power, coefficient) in enumerate(self.coefficients)
                    if coefficient != 0]
        return "$ %s $" % (" + ".join(monomials))
    @classmethod
    def zero(cls):
        return Polynomial(0)
```

Базис для множества всех полиномов — это бесконечное множество $\{1, x, x^2, x^3, x^4...\}$. Учитывая доступность всех возможных степеней x, можно построить любой полином как линейную комбинацию.

Упражнение 6.38. Ранее в этом разделе я показал псевдокод, конструирующий первый базисный вектор 270 000-мерного пространства изображений. Как будет выглядеть второй базисный вектор?

Решение. Второй базисный вектор можно получить, поставив единицу на следующее возможное место. Это даст тусклый зеленый пиксел в верхнем левом углу изображения:

```
ImageVector([
    (0,1,0), (0,0,0), (0,0,0), ..., (0,0,0),
    (0,0,0), (0,0,0), (0,0,0), ..., (0,0,0),
    ...
])
```

Второй базисный вектор, 1 переместилась на вторую возможную позицию

Все остальные содержат черные пикселы

Упражнение 6.39. Напишите функцию `solid_color(r, g, b)`, которая возвращает `ImageVector` сплошного цвета с заданным значением красного, зеленого и синего компонентов во всех пикселах.

Решение

```
def solid_color(r,g,b):
    return ImageVector([(r,g,b) for _ in range(0,300*300)])
```

Упражнение 6.40. Мини-проект. Напишите линейное отображение, генерирующее `ImageVector` из черно-белого изображения 30×30, реализованного в виде матрицы 30×30 значений яркости. Затем реализуйте линейное отображение, которое преобразует изображение 300×300 в черно-белое изображение 30×30, усредняя яркость (среднее значение красного, зеленого и синего компонентов) в каждом пикселе.

Решение

```
image_size = (300,300)
total_pixels = image_size[0] * image_size[1]
square_count = 30
square_width = 10

def ij(n):
    return (n // image_size[0], n % image_size[1])

def to_lowres_grayscale(img):
    matrix = [
        [0 for i in range(0,square_count)]
```

Указывает, что изображение разбивается по сетке 30×30

Функция принимает ImageVector и возвращает массив с 30 массивами по 30 значений оттенков серого цвета в каждом

```
        for j in range(0,square_count)
    ]
    for (n,p) in enumerate(img.pixels):
        i,j = ij(n)
        weight = 1.0 / (3 * square_width * square_width)
        matrix[i // square_width][ j // square_width] += (sum(p) * weight)
    return matrix

def from_lowres_grayscale(matrix):
    def lowres(pixels, ij):
        i,j = ij
        return pixels[i // square_width][ j // square_width]
    def make_highres(limg):
        pixels = list(matrix)
        triple = lambda x: (x,x,x)
        return ImageVector([triple(lowres(matrix, ij(n)))
                            for n in range(0,total_pixels)])
    return make_highres(matrix)
```

Вторая функция принимает матрицу 30 × 30 и возвращает изображение, построенное из блоков 10 × 10 пикселов, с яркостью, заданной значениями матрицы

Вызов `from_lowres_grayscale(to_lowres_grayscale(img))` преобразует изображение `img` так, как я показал ранее в этой главе.

КРАТКИЕ ИТОГИ ГЛАВЫ

- Векторное пространство — это обобщение двухмерной плоскости и трехмерного пространства — совокупность объектов, которые можно складывать и умножать на скаляры. Операции сложения и умножения на скаляр должны подчиняться определенным правилам (перечислены в разделе 6.1.5), чтобы моделировать более знакомые операции в двух- и трехмерном пространствах.
- На языке Python обобщение можно выразить, выделив общие черты разных типов данных в абстрактный базовый класс, и наследовать его в определениях других типов.
- Python поддерживает перегрузку арифметических операторов, что позволяет единообразно записывать векторные вычисления в коде независимо от используемых векторов.
- Сложение и умножение на скаляр должны подчиняться определенным правилам, чтобы мы могли понять эти действия, и их можно проверить, написав модульные тесты со случайными векторами.
- Объекты реального мира, такие как подержанные автомобили, можно описать несколькими числами (координатами) и, следовательно, рассматривать как векторы. Это позволяет оперировать такими абстрактными понятиями, как среднее двух автомобилей.

- Функции можно рассматривать как векторы. Их можно складывать и перемножать, складывая и перемножая выражения, которые их определяют.
- Матрицы можно рассматривать как векторы. Элементы матрицы $m \times n$ можно считать координатами $(m \cdot n)$-мерного вектора. Сложение или умножение матриц на скаляр дает тот же эффект, что и сложение или умножение на скаляр определяемых ими линейных функций.
- Изображения фиксированной высоты и ширины образуют векторное пространство. Они определяются значением красного, зеленого и синего компонентов в каждом пикселе, поэтому количество координат и, следовательно, размерность пространства определяется утроенным числом пикселов.
- Подпространство векторного пространства — это подмножество векторов в векторном пространстве, которое само является векторным пространством. То есть никакие линейные комбинации векторов в подпространстве не выходят за пределы этого подпространства.
- Для любой прямой, проходящей через начало координат в двух- или трехмерном пространстве, лежащие на ней векторы образуют одномерное подпространство. Для любой плоскости, проходящей через начало координат в трехмерном пространстве, лежащие на ней векторы образуют двухмерное подпространство.
- Линейная оболочка (span) набора векторов — это совокупность всех их линейных комбинаций. Она гарантированно является подпространством любого пространства, в котором находятся векторы.
- Набор векторов считается линейно *независимым*, если ни один из них нельзя получить как линейную комбинацию других. В противном случае множество линейно *зависимо*. Набор линейно независимых векторов, охватывающих векторное пространство (или подпространство), называется *базисом* этого пространства. Для данного пространства любой базис будет иметь одинаковое количество векторов. Это число определяет размерность пространства.
- Когда есть возможность интерпретировать данные как находящиеся в векторном пространстве, часто оказывается, что подпространства состоят из данных со схожими свойствами. Например, подмножество векторов, образующих подпространство изображений со сплошными цветами.

7 Решение систем линейных уравнений

В этой главе

- ✓ Обнаружение столкновений объектов в двухмерной видеоигре.
- ✓ Запись уравнений для представления линий и определение точек пересечения линий на плоскости.
- ✓ Изображение и решение систем линейных уравнений в пространствах с тремя и большим числом измерений.
- ✓ Запись векторов в виде линейных комбинаций других векторов.

Размышляя об алгебре, вы, вероятно, представляете себе задачи, которые требуют найти x. Возможно, на уроках алгебры вы потратили много времени, изучая приемы решения таких уравнений, как $3x^2 + 2x + 4 = 0$, то есть выяснения того, какое значение или значения x делают уравнение верным.

Линейная алгебра как один из разделов алгебры подразумевает решение аналогичных задач. Разница лишь в том, что решением может быть вектор или матрица, а не число. Пройдя традиционный курс линейной алгебры, можно изучить множество алгоритмов решения подобных задач. Но поскольку в нашем распоряжении есть Python, нам достаточно знать, как определить задачу и выбрать правильную библиотеку для ее решения.

Далее я расскажу о наиболее важном классе задач линейной алгебры, которые вы встретите в своей практике, — *системах линейных уравнений*. Эти задачи сводятся к поиску точек пересечения линий, плоскостей или их многомерных аналогов. Один из примеров — известная школьная математическая задача о двух поездах, отправляющихся из пункта А и пункта Б в разное время и с разной скоростью. Но я не думаю, что вас интересует работа железных дорог, поэтому приведу более занимательный пример.

В этой главе мы создадим простой ремейк классической аркадной игры «Астероиды» (рис. 7.1). В ней игрок управляет треугольником, представляющим космический корабль, и стреляет из лазерной пушки по плавающим вокруг него многоугольникам, которые изображают астероиды. Игрок должен уничтожить астероиды, чтобы они не столкнулись с космическим кораблем и не уничтожили его.

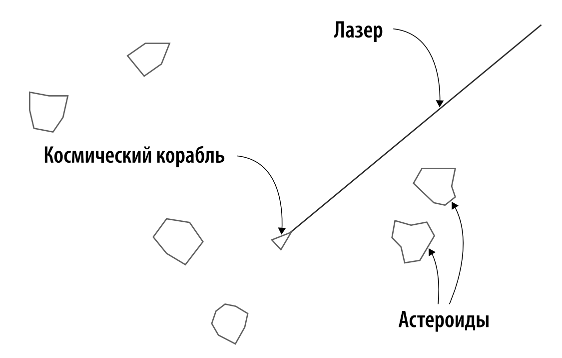


Рис. 7.1. Классическая аркадная игра «Астероиды»

Одна из ключевых особенностей этой игры — определение попадания луча лазера в астероид. Для этого мы должны выяснить, пересекает ли линия, представляющая лазерный луч, отрезки, ограничивающие астероиды. Если линии пересекаются, то астероид уничтожается. Сначала подготовим игровое поле, а затем посмотрим, как решается базовая задача, относящаяся к линейной алгебре.

После реализации игры я покажу, как этот двухмерный пример обобщается на три и любое другое количество измерений. Вторая половина главы в основном будет посвящена теории и завершит наше знакомство с линейной алгеброй. Мы рассмотрели множество базовых понятий, которые можно найти в учебниках по алгебре для средней школы, хотя и не так подробно, как там. По завершении этой главы вы будете хорошо подготовлены к тому, чтобы засесть за более подробный учебник по линейной алгебре и восполнить недостающие детали. А пока сосредоточимся на создании игры.

7.1. РАЗРАБОТКА АРКАДНОЙ ИГРЫ

В этой главе я покажу упрощенную версию игры с астероидами, в которой корабль и астероиды статичны. В примерах исходного кода, сопровождающих книгу, вы найдете версию с движущимися астероидами, а в части II книги я расскажу, как заставить их двигаться в соответствии с законами физики. Для начала смоделируем игровые объекты — космический корабль, лазер и астероиды — и посмотрим, как можно отобразить их на экране.

7.1.1. Моделирование игры

В этом разделе мы изобразим космический корабль и астероиды в виде многоугольников. Как и прежде, смоделируем их в виде набора векторов. Например, восьмиугольный астероид можно представить восемью векторами (обозначены стрелками на рис. 7.2) и соединить их отрезками, чтобы нарисовать контур.

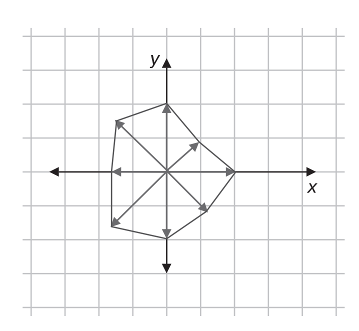


Рис. 7.2. Восьмиугольник, представляющий астероид

Астероид или космический корабль движется или вращается, перемещаясь в космосе, но его форма остается неизменной. Поэтому представляющие ее векторы будут храниться отдельно от координат x и y его центра, которые могут меняться со временем. Мы также сохраним отдельно угол, определяющий угол поворота объекта в текущий момент. Класс `PolygonModel` представляет игровой объект (корабль или астероид), хранящий его форму и способный перемещаться или вращаться. Каждый экземпляр класса инициализируется набором векторов (точек), определяющих контур, а также координатами центра x и y и углом поворота, равными нулю:

```
class PolygonModel():
    def __init__(self,points):
        self.points = points
        self.rotation_angle = 0
        self.x = 0
        self.y = 0
```

Чтобы узнать фактическое местоположение движущегося космического корабля или астероида, нужно применить к нему операцию параллельного переноса по `self.x`, `self.y` и операцию поворота на угол `self.rotation_angle`. В качестве упражнения можете попробовать реализовать в классе `PolygonModel` метод вычисления фактических векторов, получающихся после преобразования.

Космический корабль и астероиды — это особые случаи `PolygonModel`, которые автоматически инициализируются соответствующими формами. Например, корабль имеет фиксированную треугольную форму, заданную тремя точками:

```
class Ship(PolygonModel):
    def __init__(self):
        super().__init__([(0.5,0), (-0.25,0.25), (-0.25,-0.25)])
```

Астероиды инициализируются 5–9 векторами с равными углами и случайными длинами от 0,5 до 1,0. Эта случайность придает астероидам некоторую уникальность:

```
class Asteroid(PolygonModel):
    def __init__(self):
        sides = randint(5,9)          ◄── Для каждого астероида выбирается
                                          случайное число сторон от 5 до 9
        vs = [vectors.to_cartesian((uniform(0.5,1.0), 2*pi*i/sides))
                for i in range(0,sides)]   ◄── Длины векторов тоже выбираются
                                               случайно в диапазоне от 0,5 до 1,0, а углы
                                               кратны 2π/n, где n — количество сторон
        super().__init__(vs)
```

Определив эти объекты, можно заняться их созданием и отображением на экране.

7.1.2. Отображение игрового поля

Чтобы получить начальное состояние игры, нам понадобятся корабль и несколько астероидов. Корабль может находиться в центре экрана, но астероиды должны быть случайным образом разбросаны по игровому полю. Будем считать, что игровая плоскость простирается от -10 до 10 вдоль осей x и y:

```
ship = Ship()

asteroid_count = 10
asteroids = [Asteroid() for _ in range(0,asteroid_count)]   ◄── Создать список с заданным числом
                                                                 объектов-астероидов, в данном случае 10

for ast in asteroids:          ◄── Задать случайные начальные координаты
    ast.x = randint(-9,9)          для каждого объекта в диапазоне от –10 до 10,
    ast.y = randint(-9,9)          чтобы все они оказались в пределах игрового поля
```

Я взял экран с разрешением 400×400 пикселов, поэтому перед отображением объектов необходимо выполнить преобразование координат x и y. При использовании двухмерной графики на основе PyGame вместо OpenGL верхний левый пиксел на экране имеет координаты (0, 0), а правый нижний — координаты (400, 400). Мало того что эти координаты больше, они также смещены и перевернуты, поэтому нужно написать функцию `to_pixels` (принцип ее действия показан на рис. 7.3), которая выполнит преобразование из нашей системы координат в пикселы PyGame.

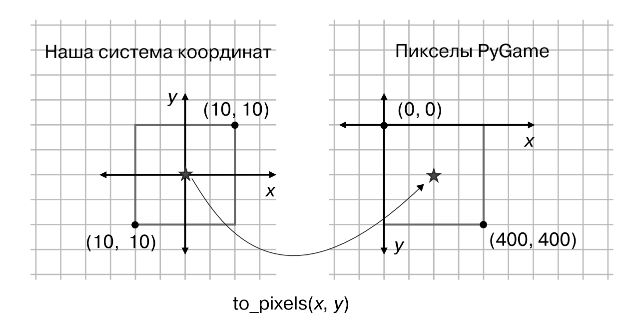


Рис. 7.3. Функция to_pixels отображает объект из центра нашей системы координат в центр экрана PyGame

Имея функцию `to_pixels`, можно написать функцию для рисования многоугольника, определяемого точками, на экране PyGame. Она получает точки (перемещенные и повернутые), определяющие многоугольник, и преобразует их в пикселы. Затем эти точки соединяются отрезками с помощью функции PyGame:

```
GREEN = (0, 255, 0)
def draw_poly(screen, polygon_model, color=GREEN):
    pixel_points = [to_pixels(x,y) for x,y in polygon_model.transformed()]
    pygame.draw.aalines(screen, color, True, pixel_points, 10)
```

Рисует отрезки, соединяющие заданные точки, которые составляют указанный объект. Параметр True означает, что будет отрисован отрезок, соединяющий первую и последнюю точки, чтобы получился замкнутый многоугольник

Полную реализацию игрового цикла можно найти в примерах исходного кода. Он фактически вызывает `draw_poly` для рисования корабля и каждого астероида при отображении каждого кадра. В результате в окне PyGame отображается игровая сцена с треугольным космическим кораблем, окруженным полем астероидов (рис. 7.4).



Рис. 7.4. Вид игровой сцены в окне PyGame

7.1.3. Стрельба из лазерной пушки

Теперь перейдем к самой важной части — дадим нашему кораблю возможность защищаться! Игрок должен иметь возможность наводить корабль с помощью клавиш со стрелками влево и вправо и стрелять из лазерной пушки, нажимая пробел. Лазерный луч должен исходить из носовой части космического корабля и достигать края экрана.

В придуманном нами двухмерном мире лазерный луч представляет собой отрезок, начинающийся в точке в носу корабля и простирающийся в том направлении, куда нацелен корабль. Мы можем заставить его достичь края экрана, сделав достаточно длинным. Поскольку отрезок, представляющий луч лазера, связан с положением объекта `Ship`, создадим метод для его вычисления в классе `Ship`:

```
class Ship(PolygonModel):
    ...
    def laser_segment(self):
        dist = 20. * sqrt(2)
        x,y = self.transformed()[0]
        return ((x,y),
                (x + dist * cos(self.rotation_angle),
                 y + dist*sin(self.rotation_angle)))
```

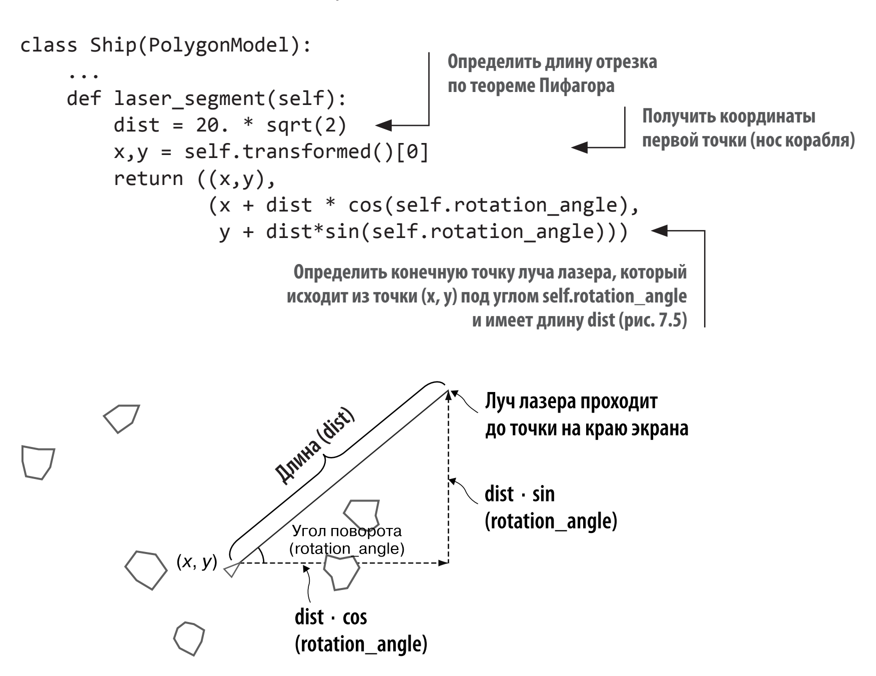


Рис. 7.5. Определение конечной точки луча лазера на краю экрана с использованием тригонометрии

В примерах исходного кода можете увидеть, как заставить PyGame реагировать на нажатия клавиш и рисовать луч лазера, только когда нажата клавиша пробела. Наконец, стреляя из лазерной пушки, игрок может попасть в астероид, и мы должны определить это. В каждой итерации игрового цикла мы должны проверить каждый астероид — не был ли он поражен лазером в данный момент. Делаться это будет с помощью метода `dos_intersect(segment)` класса `PolygonModel`, который вычисляет, пересекает ли входной отрезок `segment`

какой-либо отрезок данного многоугольника `PolygonModel`. Заключительный код содержит следующие несколько строк:

```
laser = ship.laser_segment()
keys = pygame.key.get_pressed()
    if keys[pygame.K_SPACE]:
    draw_segment(*laser)

    for asteroid in asteroids:
        if asteroid.does_intersect(laser):
            asteroids.remove(asteroid)
```

Вычислить отрезок, представляющий лазерный луч, на основе текущего положения и ориентации корабля

Определить, какие клавиши нажаты. Если нажата клавиша пробела, то нарисовать лазерный луч с помощью вспомогательной функции draw_segment, похожей на draw_poly

Для каждого астероида проверить, не пересекает ли луч лазера какой-нибудь из его отрезков. Если пересекает, то уничтожить данный астероид, удалив его из списка астероидов asteroids

Осталось реализовать метод `do_intersect(segment)`. В следующем разделе мы рассмотрим математические основы, необходимые для этого.

7.1.4. Упражнения

Упражнение 7.1. Реализуйте метод `transform()` в `PolygonModel`, который возвращает точки модели после параллельного переноса на вектор, определяемый атрибутами x и y, и поворота на угол `rotate_angle`.

Решение. Сначала нужно выполнить поворот, иначе вектор переноса также повернется на угол `rotate_angle`, например:

```
class PolygonModel():
    ...
    def transformed(self):
        rotated = [vectors.rotate2d(self.rotation_angle, v)
                   for v in self.points]
        return [vectors.add((self.x,self.y),v) for v in rotated]
```

Упражнение 7.2. Напишите функцию `to_pixels(x, y)`, которая принимает пару координат x и y, где $-10 < x < 10$ и $-10 < y < 10$, и отображает их в соответствующие координаты PyGame x и y, каждая из которых изменяется в диапазоне от 0 до 400.

Решение

```
width, height = 400, 400
def to_pixels(x,y):
    return (width/2 + width * x / 20, height/2 - height * y / 20)
```

7.2. ОПРЕДЕЛЕНИЕ ТОЧЕК ПЕРЕСЕЧЕНИЯ ЛИНИЙ

Наша задача — определить, попадает ли луч лазера в астероид. Для этого мы рассмотрим каждый отрезок в контуре астероида и определим, пересекается ли он с отрезком, представляющим луч лазера. Для этого можно использовать несколько алгоритмов, я выбрал основанный на решении *системы линейных уравнений с двумя переменными*. Геометрически это означает нахождение точки пересечения прямых, определяющих край астероида и луч лазера (рис. 7.6).

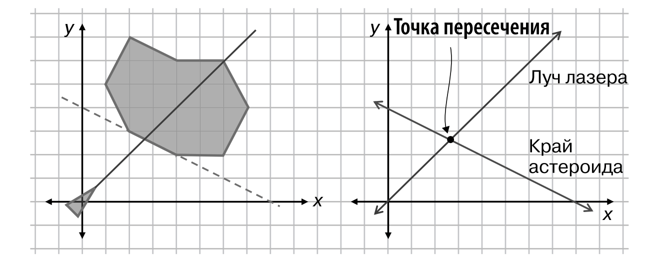


Рис. 7.6. Луч лазера, поражающий край астероида *(слева)*, и соответствующая система линейных уравнений *(справа)*

Определив координаты точки пересечения прямых, можно проверить, находится ли она в пределах обоих отрезков. Если да, значит, отрезки пересекаются и луч лазера попадает в цель. Сначала рассмотрим уравнения прямых на плоскости, затем займемся решением задачи определения координат точки пересечения пар прямых, а в заключение реализуем для нашей игры метод `does_intersect`.

7.2.1. Выбор правильной формулы прямой

В предыдущей главе мы видели, что одномерные подпространства на двухмерной плоскости являются прямыми линиями. Эти подпространства состоят из всех произведений на скаляр $t \cdot \mathbf{v}$ для выбранного вектора $\mathbf{v}$. Поскольку одно из произведений — это $0 \cdot \mathbf{v}$, все такие прямые проходят через начало координат, поэтому $t \cdot \mathbf{v}$ нельзя назвать общей формулой для любых прямых.

Если допустить возможность параллельного переноса прямой, проходящей через начало координат, на вектор $\mathbf{u}$, то можно получить любую возможную прямую. Точки на этой прямой будут определяться формулой $\mathbf{u} + t \cdot \mathbf{v}$ для некоторого скаляра t. Например, возьмем $\mathbf{v} = (2, -1)$. Точки вида $t \cdot (2, -1)$ лежат на прямой, проходящей через начало координат. Но если выполнить перенос на вектор $\mathbf{u} = (2, 3)$, то теперь точки будут определяться формулой $(2, 3) + t \cdot (2, -1)$ и лежать на прямой, *не* проходящей через начало координат (рис. 7.7).

Любую прямую можно описать как совокупность точек $\mathbf{u} + t \cdot \mathbf{v}$ для некоторого набора векторов $\mathbf{u}$ и $\mathbf{v}$ и *всех* возможных скалярных множителей t. Вероятно,

это не та универсальная формула прямой, к которой вы привыкли. Вместо того чтобы записывать y как функцию от x, мы задали координаты x и y точек на прямой как функцию от t. Иногда можно увидеть формулу прямой $\mathbf{r}(t) = \mathbf{u} + t \cdot \mathbf{v}$. Такая запись указывает, что эта прямая является векторнозначной функцией $\mathbf{r}$ скалярного параметра t. Параметр t определяет, сколько единиц $\mathbf{v}$ нужно пройти от начальной точки $\mathbf{u}$, чтобы получить результат $\mathbf{r}(t)$.

Преимущество этой формулы прямой в том, что ее легко найти, имея две точки. Если точками являются $\mathbf{u}$ и $\mathbf{w}$, то можно использовать $\mathbf{u}$ в качестве вектора переноса, а $\mathbf{w} - \mathbf{u}$ — в качестве масштабируемого вектора (рис. 7.8).

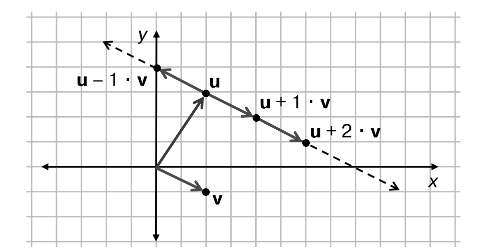


Рис. 7.7. Векторы **u** = (2, 3) и **v** = (2, –1). Точки вида **u** + t · **v** лежат на прямой

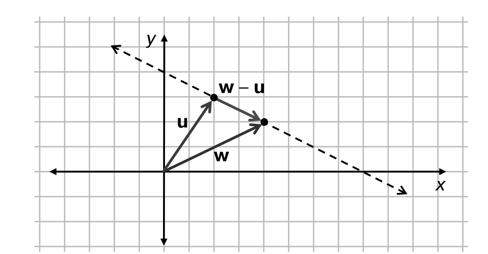


Рис. 7.8. Для заданных **u** и **w** соединяющая их прямая определяется как **r**(t) = **u** + t · (**w** – **u**)

Формула $\mathbf{r}(t) = \mathbf{u} + t \cdot \mathbf{v}$ также имеет обратную сторону. Как вы увидите в упражнениях, в этой форме можно записать одну и ту же прямую несколькими способами. Кроме того, дополнительный параметр t усложняет решение уравнений, потому что появляется одна дополнительная неизвестная переменная. Поэтому я предлагаю взглянуть на альтернативные формулы, обладающие своими преимуществами.

В средней школе мы выучили формулу прямой $y = m \cdot x + b$. Она удобна тем, что явно выражает координату y как функцию координаты x. Эта форма позволяет легко начертить прямую: нужно взять несколько значений x, вычислить соответствующие им значения y и нарисовать точки с полученными координатами (x, y). Но эта формула имеет и некоторые ограничения. Наиболее существенное таково: она не позволяет представить вертикальную прямую, такую как $\mathbf{r}(t) = (3, 0) + t \cdot (0, 1)$, состоящую из векторов с $x = 3$.

Мы продолжим использовать *параметрическую* формулу $\mathbf{r}(t) = \mathbf{u} + t \cdot \mathbf{v}$, потому что она избавлена от таких проблем, но было бы здорово иметь формулу без дополнительного параметра t, способную представлять любую прямую. Одна из таких формул имеет вид $ax + by = c$. Например, прямую, которую мы видели на нескольких последних рисунках, можно записать как $x + 2y = 8$ (рис. 7.9). Это совокупность точек (x, y) на плоскости, удовлетворяющих этому уравнению.

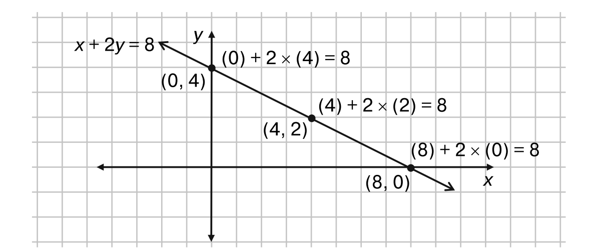


Рис. 7.9. Координаты (x, y) всех точек на прямой удовлетворяют уравнению $x + 2y = 8$

Формула $ax + by = c$ не имеет дополнительных параметров и позволяет выразить любую прямую, даже вертикальную. Например, $x = 3$ — это то же самое, что $1 \cdot x + 0 \cdot y = 3$. Любое уравнение, представляющее прямую, называется *линейным*, а эта формула, в частности, называется *канонической формой* линейного уравнения. Мы продолжим использовать ее в этой главе, потому что она поможет упростить организацию вычислений.

7.2.2. Поиск стандартной формы уравнения прямой

Формула $x + 2y = 8$ — это уравнение прямой, содержащей один из отрезков контура астероида в нашем примере. Далее рассмотрим еще один отрезок (рис. 7.10), а затем попробуем систематизировать определение стандартной формы линейных уравнений. Приготовьтесь к погружению в алгебру! Я подробно объясню каждый шаг, но кому-то эти объяснения могут показаться суховатыми. Будет намного лучше, если вы станете следить за ними, вооружившись карандашом и бумагой.

Вектор $(1, 5) - (2, 3)$ равен $(-1, 2)$, который параллелен прямой. Поскольку $(2, 3)$ лежит на прямой, параметрическое уравнение для нее имеет вид $r(t) = (2, 3) + t \cdot (-1, 2)$. Зная, что все точки на прямой определяются формулой $(2, 3) + t \cdot (-1, 2)$ для некоторого t, можно ли выразить это условие в виде уравнения стандартной формы? Нам нужно заняться алгебраическими вычислениями и, в частности, избавиться от t. Поскольку $(x, y) = (2, 3) + t \cdot (-1, 2)$, мы фактически имеем два уравнения:

$$x = 2 - t;$$
$$y = 3 + 2t.$$

Можем манипулировать ими обоими, чтобы получить два новых уравнения с одинаковыми значениями $2t$:

$$4 - 2x = 2t;$$
$$y - 3 = 2t.$$

Оба выражения слева от знака равенства равны $2t$, а значит, они равны друг другу:

$$4 - 2x = y - 3.$$

Мы избавились от t! Наконец, перенеся члены x и y на одну сторону, получаем уравнение стандартной формы:

$$2x + y = 7.$$

Процесс не особенно сложный, но нам нужно знать, как воплотить его в программном коде. Попробуем решить общую задачу: даны две точки, (x_1, y_1) и (x_2, y_2), как будет выглядеть уравнение прямой, проходящей через них (рис. 7.11)?

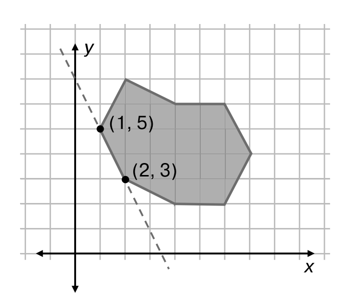


Рис. 7.10. Точки (1, 5) и (2, 3) определяют второй отрезок контура астероида

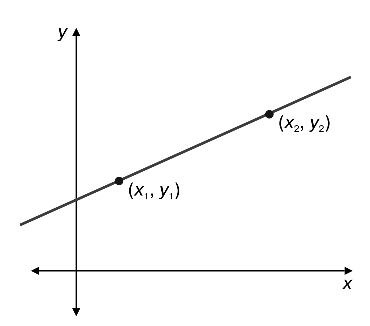


Рис. 7.11. Общая задача определения уравнения прямой, проходящей через две известные точки

Согласно параметрической формуле точки на прямой имеют следующий вид:

$$(x, y) = (x_1, y_1) + t \cdot (x_2 - x_1, y_2 - y_1).$$

Здесь множество переменных x и y, но не забывайте, что x_1, x_2, y_1 и y_2 в данном контексте являются константами. Если бы у нас были две точки с известными координатами, мы с таким же успехом могли бы назвать их (a, b) и (c, d). Настоящие переменные — это x и y (без индексов), которые обозначают координаты любой точки на линии. Как и прежде, разобьем это уравнение на две части:

$$x = x_1 + t \cdot (x_2 - x_1);$$
$$y = y_1 + t \cdot (y_2 - y_1).$$

Перенесем x_1 и y_1 в левые части соответствующих уравнений:

$$x - x_1 = t \cdot (x_2 - x_1);$$
$$y - y_1 = t \cdot (y_2 - y_1).$$

Наша следующая цель — преобразовать уравнения так, чтобы их правые части выглядели одинаково и появилась возможность уравнять левые части. Умножив обе части первого уравнения на $(y_2 - y_1)$ и обе части второго уравнения на $(x_2 - x_1)$, получаем:

$$(y_2 - y_1) \cdot (x - x_1) = t \cdot (x_2 - x_1) \cdot (y_2 - y_1);$$
$$(x_2 - x_1) \cdot (y - y_1) = t \cdot (x_2 - x_1) \cdot (y_2 - y_1).$$

Поскольку правые части идентичны, можно с уверенностью утверждать, что и левые части первого и второго уравнений равны друг другу. Это позволяет записать новое уравнение без t:

$$(y_2 - y_1) \cdot (x - x_1) = (x_2 - x_1) \cdot (y - y_1).$$

Как вы наверняка помните, наша цель — получить уравнение вида $ax + by = c$, поэтому перенесем x и y влево, а константы — вправо. Для начала раскроем скобки с обеих сторон:

$$(y_2 - y_1) \cdot x - (y_2 - y_1) \cdot x_1 = (x_2 - x_1) \cdot y - (x_2 - x_1) \cdot y_1.$$

Теперь перенесем константы в правую часть, а переменные — в левую:

$$(y_2 - y_1) \cdot x - (x_2 - x_1) \cdot y = (y_2 - y_1) \cdot x_1 - (x_2 - x_1) \cdot y_1.$$

Раскроем скобки в правой части и сократим подобные члены:

$$(y_2 - y_1) \cdot x - (x_2 - x_1) \cdot y = y_2x_1 - y_1x_1 - x_2y_1 + x_1y_1 = x_1y_2 - x_2y_1.$$

Вот и все! Мы получили линейное уравнение в стандартной форме $ax + by = c$, где $a = (y_2 - y_1)$, $b = -(x_2 - x_1)$, или, иначе, $b = (x_1 - x_2)$, а $c = (x_1y_2 - x_2y_1)$. Проверим его на предыдущем примере, использовав две точки, $(x_1, y_1) = (2, 3)$ и $(x_2, y_2) = (1, 5)$:

$$a = y_2 - y_1 = 5 - 3 = 2;$$
$$b = -(x_2 - x_1) = -(1 - 2) = 1;$$
$$c = x_1y_2 - x_2y_1 = 2 \cdot 5 - 3 \cdot 1 = 7.$$

Это означает, что уравнение в стандартной форме имеет вид $2x + y = 7$. Формула явно заслуживает доверия! В качестве последнего примера найдем уравнение в стандартной форме, описывающее прямую, представляющую луч лазера. Судя по тому, как я нарисовал ее на рис. 7.6 и как снова показано на рис. 7.12, она проходит через точки (2, 2) и (4, 4).

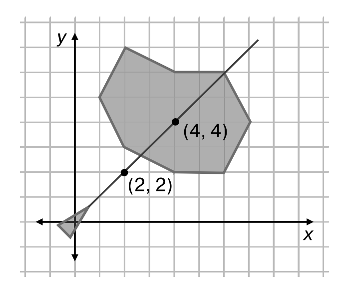


Рис. 7.12. Луч лазера проходит через точки (2, 2) и (4, 4)

В нашей игре имеются начальная и конечная точки луча лазера, но для примера подойдут и эти числа. Подставив их в формулу, находим:

$$a = y_2 - y_1 = 4 - 2 = 2;$$
$$b = -(x_2 - x_1) = -(4 - 2) = -2;$$
$$c = x_1 y_2 - x_2 y_1 = 2 \cdot 4 - 2 \cdot 4 = 0.$$

Уравнение $2y - 2x = 0$ эквивалентно уравнению $x - y = 0$ или просто $x = y$. Чтобы определить, попадает ли лазер в астероид, нужно найти, где прямая $x - y = 0$ пересекается с прямой $x + 2y = 8$, $2x + y = 7$ или любой другой прямой, ограничивающей астероид.

7.2.3. Линейные уравнения в матричной записи

Сосредоточимся на пересечении, которое видно явно: лазер попадает в ближайший край астероида — отрезок, лежащий на прямой $x + 2y = 8$ (рис. 7.13).

Выполнив некоторые изыскания, мы получаем свою первую настоящую систему линейных уравнений. Системы линейных уравнений принято записывать в виде сетки, как показано далее, чтобы переменные x и y находились друг под другом:

$$x - y = 0;$$
$$x + 2y = 8.$$

Как говорилось в главе 5, эти два уравнения можно объединить в одно матричное уравнение. Сделать это можно несколькими способами. Один из них — записать линейную комбинацию векторов-столбцов, где x и y — коэффициенты:

$$x\begin{pmatrix}1\\1\end{pmatrix} + y\begin{pmatrix}-1\\2\end{pmatrix} = \begin{pmatrix}0\\8\end{pmatrix}.$$

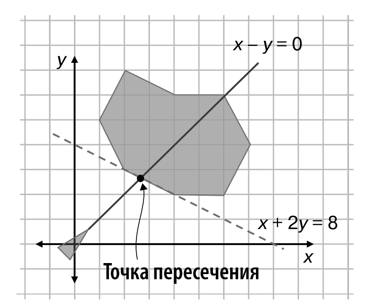


Рис. 7.13. Луч лазера попадает в астероид в точке пересечения прямых $x - y = 0$ и $x + 2y = 8$

Другой способ — записать как матричное умножение. Линейная комбинация (1, –1) и (–1, –2) с коэффициентами x и y аналогична матричному произведению

$$\begin{pmatrix} 1 & -1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ 8 \end{pmatrix}.$$

В таком виде задача решения системы линейных уравнений выглядит как определение вектора в задаче умножения матриц. Если матрице 2×2 дать имя A, то задача сводится к определению вектора (x, y), на который нужно умножить матрицу A, чтобы получить (0, 8). Иначе говоря, мы знаем, что результат линейного преобразования A — это вектор (0, 8), и нужно узнать, какие входные данные его дают (рис. 7.14).

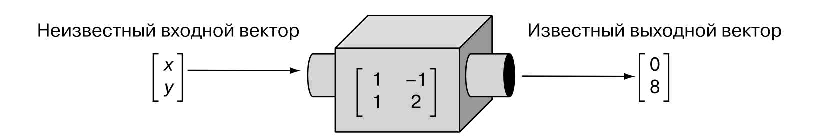


Рис. 7.14. Формулировка задачи определения входного вектора, дающего желаемый выходной вектор

Разные формы записи помогают по-новому взглянуть на одну и ту же задачу. Решение системы линейных уравнений эквивалентно определению линейной комбинации некоторых векторов, которая дает другой заданный вектор. Оно также эквивалентно определению входного вектора для линейного преобразования, которое дает заданный результат. Далее мы посмотрим, как решить все эти задачи одновременно.

7.2.4. Решение линейных уравнений с помощью NumPy

Определение точки пересечения прямых $x - y = 0$ и $x + 2y = 8$ сродни определению вектора (x, y), удовлетворяющего уравнению умножения матриц

$$\begin{pmatrix} 1 & -1 \\ 1 & 2 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ 8 \end{pmatrix}.$$

Это всего лишь разные способы записи, но постановка задачи в такой форме позволяет использовать для ее решения готовые инструменты. В частности, в библиотеке NumPy для Python имеется модуль линейной алгебры и функция, которая решает уравнения такого вида, например:

Упаковать матрицу в массив NumPy

```
>>> import numpy as np
>>> matrix = np.array(((1,-1),(1,2)))
>>> output = np.array((0,8))
>>> np.linalg.solve(matrix,output)
array([2.66666667, 2.66666667])
```

Упаковать выходной вектор в массив NumPy (его не нужно преобразовывать в вектор-столбец)

Функция numpy.linalg.solve принимает матрицу и выходной вектор и находит соответствующий входной вектор

Результат (x, y) = (2,66..., 2,66...)

Библиотека NumPy сообщила нам, что обе координаты, x и y, точки пересечения примерно равны 22/3 (или 8/3), с геометрической точки зрения это выглядит верно. Если посмотреть на график на рис. 7.13, то похоже, что обе координаты точки пересечения должны иметь значение между 2 и 3. Для большей уверенности можно проверить, лежит ли эта точка на обеих прямых, подставив ее координаты в оба уравнения:

$$1x - 1y = 1 \cdot (2{,}66666667) - 1 \cdot (2{,}66666667) = 0;$$

$$1x + 2y = 1 \cdot (2{,}66666667) + 2 \cdot (2{,}66666667) = 8{,}00000001.$$

Результаты довольно близки к (0, 8) и действительно дают точное решение. Этот вектор решения, примерно (8/3, 8/3), также является вектором, удовлетворяющим матричному уравнению

$$\begin{pmatrix} 1 & -1 \\ 1 & 2 \end{pmatrix}\begin{pmatrix} 8/3 \\ 8/3 \end{pmatrix} = \begin{pmatrix} 0 \\ 8 \end{pmatrix}.$$

Как показано на рис. 7.15, мы можем представить (8/3, 8/3) как вектор, который передается в машину линейного преобразования, определяемого матрицей, дающей нам желаемый выходной вектор.

Функцию `numpy.linalg.solve` можно рассматривать как машину другой формы, которая принимает матрицу и выходной вектор и возвращает вектор решения линейного уравнения, определяемого матрицей и выходным вектором (рис. 7.16).

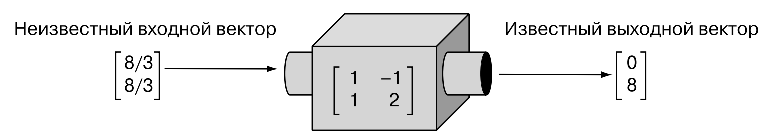


Рис. 7.15. Передача вектора (8/3, 8/3) в машину линейного преобразования дает желаемый результат (0, 8)

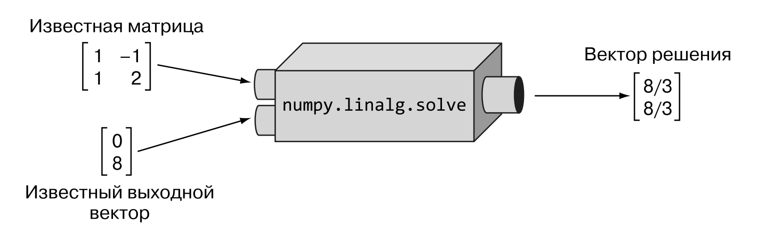


Рис. 7.16. Функция numpy.linalg.solve принимает матрицу и вектор и возвращает вектор решения заданной системы линейных уравнений

Это, пожалуй, самая важная вычислительная задача в линейной алгебре — имея матрицу A и вектор $\mathbf{w}$, найти такой вектор $\mathbf{v}$, что $A\mathbf{v} = \mathbf{w}$. Этот вектор является решением системы линейных уравнений, представленной A и $\mathbf{w}$. Нам повезло, что в языке Python есть функция, которая может сделать это самостоятельно, поэтому не нужно выполнять утомительные вычисления вручную. Теперь мы можем использовать эту функцию, чтобы узнать, поразил ли лазер астероид.

7.2.5. Определение факта попадания в астероид

Недостающей частью игры была реализация метода `does_intersect` в классе `PolygonModel`. Для любого экземпляра этого класса, представляющего объект в форме двухмерного многоугольника, данный метод должен вернуть `True`, если заданный отрезок пересекает любой из отрезков, составляющих контур объекта.

Для реализации этого метода понадобятся несколько вспомогательных функций. Во-первых, заданные отрезки нужно преобразовать из пар конечных точек в линейные уравнения в стандартной форме. В одном из упражнений в конце раздела вам будет предложено написать функцию `standard_form`, которая принимает два вектора и возвращает кортеж (a, b, c), где $ax + by = c$ — прямая, на которой лежит отрезок.

Затем, имея два отрезка, каждый из которых представлен своей парой конечных точек, нужно выяснить, где пересекаются прямые, на которых они лежат. Если u_1 и u_2 — конечные точки первого отрезка, а v_1 и v_2 — конечные точки второго

отрезка, то нужно сначала найти уравнения прямых в стандартной форме, а затем передать их в NumPy для поиска решения, например:

```
def intersection(u1,u2,v1,v2):
    a1, b1, c1 = standard_form(u1,u2)
    a2, b2, c2 = standard_form(v1,v2)
    m = np.array(((a1,b1),(a2,b2)))
    c = np.array((c1,c2))
    return np.linalg.solve(m,c)
```

Результат — это точка пересечения двух прямых, на которых лежат отрезки. Но эта точка может оказаться за пределами данных отрезков, как показано на рис. 7.17.

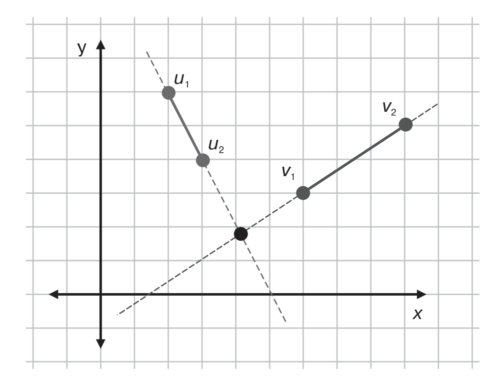


Рис. 7.17. Один отрезок соединяет точки $\mathbf{u}_1$ и $\mathbf{u}_2$, а другой — точки $\mathbf{v}_1$ и $\mathbf{v}_2$. Прямые, продолжающие отрезки, пересекаются, но сами отрезки — нет

Чтобы определить, пересекаются ли два отрезка, нужно проверить, находится ли точка пересечения прямых между двумя парами конечных точек. Выполнить такую проверку можно, используя расстояния. На рис. 7.17 точка пересечения находится дальше от точки v_2, чем точка v_1. Точно так же она находится дальше от u_1, чем u_2. Это указывает на то, что точка не лежит ни на одном из отрезков. Проверив четыре расстояния, можно точно сказать, является ли точка пересечения прямых (x, y) также точкой пересечения отрезков:

```
def do_segments_intersect(s1,s2):
    u1,u2 = s1
    v1,v2 = s2
    d1, d2 = distance(*s1), distance(*s2)   ← Сохранить длины первого и второго отрезков как d1 и d2 соответственно
    x,y = intersection(u1,u2,v1,v2)   ← Найти точку пересечения (x, y) прямых, на которых лежат отрезки
    return (distance(u1, (x,y)) <= d1 and   ← Выполнить четыре проверки, чтобы убедиться, что точка пересечения лежит на двух отрезках одновременно, то есть между четырьмя конечными точками
            distance(u2, (x,y)) <= d1 and
            distance(v1, (x,y)) <= d2 and
            distance(v2, (x,y)) <= d2)
```

Наконец, мы можем написать метод does_intersect, проверяя, возвращает ли do_segments_intersect значение True для входного отрезка и любого из ребер преобразованного многоугольника:

```
class PolygonModel():
    ...
    def does_intersect(self, other_segment):
        for segment in self.segments():
            if do_segments_intersect(other_segment,segment):
                return True
        return False
```

Если какой-то из отрезков многоугольника пересекается с other_segment, метод возвращает True

В дальнейшем при выполнении упражнений вы сможете убедиться, что этот подход действительно работает, если построите астероид и лазерный луч с известными координатами точек. Реализовав метод does_intersect так, как показано ранее, мы получаем возможность поворачивать космический корабль, целиться в астероиды и уничтожать их.

7.2.6. Определение систем без решения

Хочу сделать еще одно важное предостережение: не каждая система линейных уравнений в двухмерном мире имеет решение! В таких приложениях, как игра с астероидами, это редкость, но некоторые пары линейных уравнений могут не иметь уникальных решений или вообще не иметь решений. Если передать библиотеке NumPy систему линейных уравнений без решения, то она сгенерирует исключение, поэтому мы должны обработать эту ситуацию.

Если пара прямых на плоскости не параллельна, то они где-то пересекаются. Даже две прямые, изображенные на рис. 7.18, которые почти параллельны (почти, но не совсем), пересекаются где-то далеко-далеко.

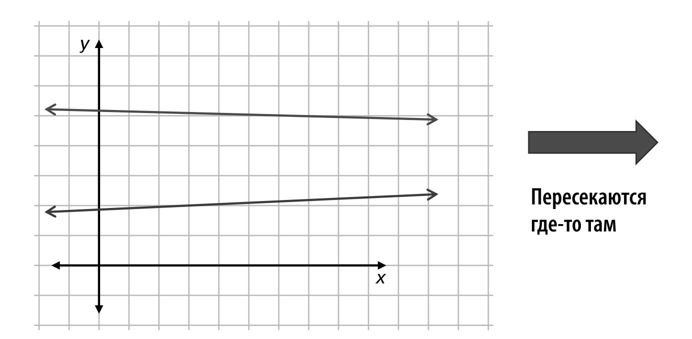


Рис. 7.18. Две почти параллельные прямые пересекаются где-то очень далеко

Проблема возникает, когда прямые оказываются параллельными, то есть когда они нигде не пересекаются (или когда отрезки лежат на одной прямой!), как показано на рис. 7.19.

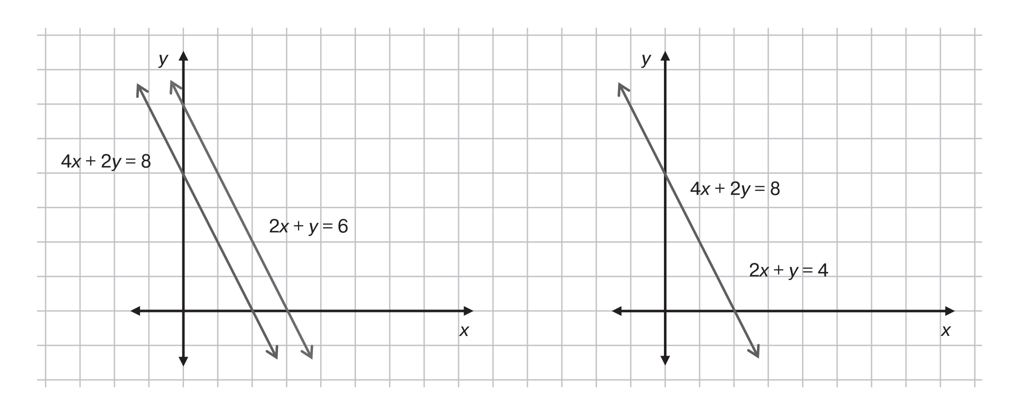


Рис. 7.19. Пара параллельных прямых, которые не пересекаются, и пара параллельных прямых, которые фактически являются одной прямой, даже при том что они определяются разными уравнениями

В первом случае у нас нет ни одной точки пересечения, а во втором — *бесконечное* количество точек пересечения: каждая точка на этих прямых — это точка пересечения. Оба этих случая представляют вычислительную проблему, потому что наш код ориентирован на получение единственного, уникального результата. Если, например, попробовать решить с помощью NumPy систему уравнений $2x + y = 6$ и $4x + 2y = 8$, то библиотека сгенерирует исключение:

```
>>> import numpy as np
>>> m = np.array(((2,1),(4,2)))
>>> v = np.array((6,4))
>>> np.linalg.solve(m,v)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
...
numpy.linalg.linalg.LinAlgError: Singular matrix
```

NumPy сообщает, что источник ошибки — это матрица. Матрица

$$\begin{pmatrix} 2 & 1 \\ 4 & 2 \end{pmatrix}$$

называется *сингулярной* в том смысле, что система линейных уравнений имеет бесконечное множество решений. Система линейных уравнений определяется матрицей и вектором, но одной матрицы уже достаточно, чтобы определить, являются ли прямые параллельными, имеет ли система единственное,

уникальное решение. Для любого ненулевого **w** в этом случае нет уникального **v**, являющегося решением системы:

$$\begin{pmatrix} 2 & 1 \\ 4 & 2 \end{pmatrix}\mathbf{v} = \mathbf{w}.$$

Мы пофилософствуем о сингулярных матрицах позже, но уже сейчас вы можете видеть, что строки (2, 1) и (4, 2) и столбцы (2, 4) и (1, 2) параллельны и, следовательно, линейно зависимы. Это ключевой признак, который говорит нам, что прямые параллельны и система не имеет единственного решения. Наличие решения для системы линейных уравнений — одно из центральных понятий линейной алгебры, оно тесно связано с понятиями линейной зависимости и размерности. Мы обсудим эту связь в двух последних разделах этой главы.

В своей игре мы можем сделать упрощающее предположение, что никакие параллельные отрезки не пересекаются. Учитывая, что игра строится на основе случайных чисел с плавающей точкой, очень маловероятно, что любые два отрезка окажутся точно параллельными. Даже если луч лазера совпадет с отрезком на границе астероида, это будет скользящее попадание, и можно считать, что оно не уничтожит астероид. Мы можем изменить `do_segments_intersect`, добавив перехват исключений, и возвращать результат по умолчанию `False`:

```
def do_segments_intersect(s1,s2):
    u1,u2 = s1
    v1,v2 = s2
    l1, l2 = distance(*s1), distance(*s2)
    try:
        x,y = intersection(u1,u2,v1,v2)
        return (distance(u1, (x,y)) <= l1 and
                distance(u2, (x,y)) <= l1 and
                distance(v1, (x,y)) <= l2 and
                distance(v2, (x,y)) <= l2)
    except np.linalg.linalg.LinAlgError:
        return False
```

7.2.7. Упражнения

Упражнение 7.3. Прямая $\mathbf{u} + t \cdot \mathbf{v}$ может проходить через начало координат. Что в таком случае можно сказать о векторах **u** и **v**?

Решение. Одна из возможностей состоит в том, что $\mathbf{u} = \mathbf{0} = (0, 0)$, в этом случае прямая автоматически проходит через начало координат. Точка $\mathbf{u} + 0 \cdot \mathbf{v}$ при этом становится началом координат независимо от координат вектора **v**. В противном случае, если **u** и **v** являются кратными друг другу, скажем, $\mathbf{u} = s \cdot \mathbf{v}$, то прямая также проходит через начало координат, потому что $\mathbf{u} - s \cdot \mathbf{v} = 0$ находится на прямой.

Упражнение 7.4. Если $\mathbf{v} = \mathbf{0} = (0, 0)$, то представляют ли точки $\mathbf{u} + t \cdot \mathbf{v}$ прямую?

Решение. Нет, независимо от значения t имеем $\mathbf{u} + t \cdot \mathbf{v} = \mathbf{u} + t \cdot (0, 0) = \mathbf{u}$. Любая точка, представленная этим уравнением, равна $\mathbf{u}$.

Упражнение 7.5. Как оказывается, формула $\mathbf{u} + t \cdot \mathbf{v}$ не единственная, то есть одну и ту же прямую можно представить разными значениями $\mathbf{u}$ и $\mathbf{v}$. Какие еще значения $\mathbf{u}$ и $\mathbf{v}$ представляют прямую $(2, 2) + t \cdot (-1, 3)$?

Решение. Один из возможных вариантов — заменить $\mathbf{v} = (-1, 3)$ его произведением на целочисленный скаляр, например $(2, -6)$. Точки вида $(2, 2) + t \cdot (-1, 3)$ совпадают с точками $(2, 2) + s \cdot (2, -6)$, где $s = -2 \cdot t$. Вектор $\mathbf{u}$ тоже можно заменить любой точкой, лежащей на прямой. Поскольку $(2, 2) + 1 \cdot (-1, 3) = (1, 5)$ лежит на прямой, уравнение $(1, 5) + s \cdot (2, -6)$ также является допустимым уравнением для той же прямой.

Упражнение 7.6. Представляет ли уравнение $a \cdot x + b \cdot y = c$ прямую при *любых* значениях a, b и c?

Решение. Нет, если a и b равны нулю, то уравнение не описывает прямую. В этом случае оно примет вид $0 \cdot x + 0 \cdot y = c$. Если $c = 0$, то это уравнение будет верно всегда, а если $c \neq 0$, то не будет верно никогда. В любом случае оно не устанавливает никакой связи между x и y и поэтому не описывает прямой.

Упражнение 7.7. Найдите другое уравнение для прямой $2x + y = 3$, показывающее, что выбор a, b и c не уникален.

Решение. Одним из таких примеров может служить уравнение $6x + 3y = 9$. В действительности, умножив обе стороны уравнения на одно и то же ненулевое число, можно получить другое уравнение для той же прямой.

Упражнение 7.8. Уравнение $ax + by = c$ эквивалентно уравнению, включающему скалярное произведение двух двухмерных векторов: $(a, b) \cdot (x, y) = c$. То есть можно сказать, что прямая — это совокупность векторов, скалярное произведение которых и заданного вектора является константой. Приведите геометрическую интерпретацию этого утверждения.

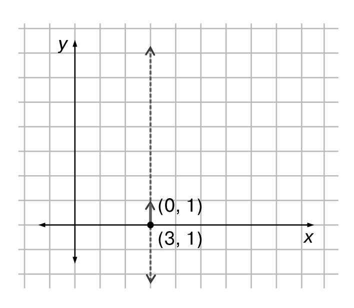
Решение. См. обсуждение в разделе 7.3.1.

Упражнение 7.9. Проверьте, удовлетворяют ли векторы (0, 7) и (3,5, 0) уравнению $2x + y = 7$.

Решение. $2 \cdot 0 + 7 = 7$ и $2 \cdot (3{,}5) + 0 = 7$.

Упражнение 7.10. Нарисуйте график прямой $(3, 0) + t \cdot (0, 1)$ и приведите эту формулу к стандартной форме.

Решение. $(3, 0) + t \cdot (0, 1)$ дает вертикальную линию с $x = 3$.



Формула $x = 3$ — это и есть уравнение прямой в стандартной форме, но мы можем подтвердить это с помощью формул. Первая точка на прямой уже задана — $(x_1, y_1) = (3, 0)$. Вторая точка на прямой — это $(3, 0) + (0, 1) = (3, 1) = (x_2, y_2)$. Отсюда имеем $a = y_2 - y_1 = 1$, $b = x_1 - x_2 = 0$ и $c = x_1y_2 - x_2y_1 = 3 \cdot 1 - 1 \cdot 0 = 3$. В результате получаем $1 \cdot x + 0 \cdot y = 3$, или просто $x = 3$.

Упражнение 7.11. Напишите на Python функцию `standard_form`, которая принимает два вектора, $\mathbf{v}_1$ и $\mathbf{v}_2$, и находит прямую $ax + by = c$, проходящую через них. В частности, функция должна вернуть кортеж констант (a, b, c).

Решение. Для этого достаточно переложить на Python наши формулы:

```
def standard_form(v1, v2):
    x1, y1 = v1
    x2, y2 = v2
    a = y2 - y1
    b = x1 - x2
    c = x1 * y2 - y1 * x2
    return a,b,c
```

Упражнение 7.12. Мини-проект. Для каждой из четырех проверок расстояния в `do_segments_intersect` найдите пару отрезков, которые не проходят эту проверку, но проходят остальные три.

Решение. Для простоты можно изменить `do_segments_intersect` так, чтобы она возвращала список значений `True`/`False` с результатами всех четырех проверок:

```
def segment_checks(s1,s2):
    u1,u2 = s1
    v1,v2 = s2
    l1, l2 = distance(*s1), distance(*s2)
    x,y = intersection(u1,u2,v1,v2)
    return [
        distance(u1, (x,y)) <= l1,
        distance(u2, (x,y)) <= l1,
        distance(v1, (x,y)) <= l2,
        distance(v2, (x,y)) <= l2
    ]
```

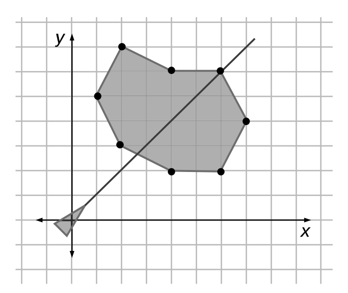
В общем случае эти проверки окажутся неудачными: один конец отрезка находится ближе к другому концу, чем к точке пересечения.

Вот несколько решений, которые я нашел, используя отрезки на линиях $y = 0$ и $x = 0$, пересекающиеся в начале координат. Каждый из них не проходит ровно одну из четырех проверок. Если вы сомневаетесь, нарисуйте их сами, чтобы увидеть и понять происходящее:

```
>>> segment_checks(((-3,0),(-1,0)),((0,-1),(0,1)))
[False, True, True, True]
>>> segment_checks(((1,0),(3,0)),((0,-1),(0,1)))
[True, False, True, True]
>>> segment_checks(((-1,0),(1,0)),((0,-3),(0,-1)))
[True, True, False, True]
>>> segment_checks(((-1,0),(1,0)),((0,1),(0,3)))
[True, True, True, False]
```

Упражнение 7.13. Для представленного примера луча лазера и астероида убедитесь, что функция `does_intersect` возвращает значение `True`.

Подсказка. Используйте линии сетки, чтобы определить координаты вершин астероида и построить объект `PolygonModel`, представляющий его.



Луч лазера попадает в астероид

Решение. В порядке против часовой стрелки, начиная с самой верхней, вершины имеют следующие координаты: (2, 7), (1, 5), (2, 3), (4, 2), (6, 2), (7, 4), (6, 6) и (4, 6). Конечными точками лазерного луча можно считать точки с координатами (1, 1) и (7, 7):

```
>>> from asteroids import PolygonModel
>>> asteroid = PolygonModel([(2,7), (1,5), (2,3), (4,2), (6,2), (7,4),
(6,6), (4,6)])
>>> asteroid.does_intersect([(0,0),(7,7)])
True
```

Этот эксперимент подтверждает, что луч лазера попал в астероид! Напротив, выстрел вертикально вверх по оси y из точки (0, 0) в точку (0, 7) идентифицируется как промах:

```
>>> asteroid.does_intersect([(0,0),(0,7)])
False
```

Упражнение 7.14. Напишите метод `does_collide(other_polygon)`, определяющий столкновение объекта `PolygonModel` с другим объектом `other_polygon`, проверяя пересечение каких-либо отрезков, составляющих их контуры. Это может помочь определить момент столкновения астероида с кораблем или другим астероидом.

Решение. Для начала добавим в `PolygonModel` вспомогательный метод `segments()`, чтобы не повторять код, возвращающий преобразованные

отрезки, которые составляют многоугольник. Затем с помощью `does_intersect` проверим каждый отрезок другого многоугольника — не пересекает ли он любой из отрезков основного многоугольника:

```
class PolygonModel():
    ...
    def segments(self):
        point_count = len(self.points)
        points = self.transformed()
        return [(points[i], points[(i+1)%point_count])
                for i in range(0,point_count)]

    def does_collide(self, other_poly):
        for other_segment in other_poly.segments():
            if self.does_intersect(other_segment):
                return True
        return False
```

Чтобы проверить новый метод, определим несколько квадратов, которые должны перекрываться и не должны перекрываться, и посмотрим, правильно ли метод `does_collide` определяет разные ситуации:

```
>>> square1 = PolygonModel([(0,0), (3,0), (3,3), (0,3)])
>>> square2 = PolygonModel([(1,1), (4,1), (4,4), (1,4)])
>>> square1.does_collide(square2)
True
>>> square3 = PolygonModel([(-3,-3),(-2,-3),(-2,-2),(-3,-2)])
>>> square1.does_collide(square3)
False
```

Упражнение 7.15. Мини-проект. Мы не можем выбрать вектор **w** так, чтобы следующая система имела единственное решение **v**:

$$\begin{pmatrix} 2 & 1 \\ 4 & 2 \end{pmatrix} \mathbf{v} = \mathbf{w}.$$

Найдите вектор **w**, такой, для которого существует *бесконечно* много решений системы, то есть бесконечно много значений **v**, удовлетворяющих уравнению.

Решение. Если $\mathbf{w} = (0, 0)$, например, то две прямые, представленные системой, идентичны. (Нарисуйте их, если не верите!) Решения имеют вид $\mathbf{v} = (a, -2a)$ для любого действительного числа a. Вот некоторые из бесконечного числа вариантов **v**, когда $\mathbf{w} = (0, 0)$:

$$\begin{pmatrix} 2 & 1 \\ 4 & 2 \end{pmatrix}\begin{pmatrix} 1 \\ -2 \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 4 & 2 \end{pmatrix}\begin{pmatrix} -4 \\ 8 \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 4 & 2 \end{pmatrix}\begin{pmatrix} 10 \\ -20 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

7.3. ОБОБЩЕНИЕ ЛИНЕЙНЫХ УРАВНЕНИЙ НА БОЛЬШЕЕ ЧИСЛО ИЗМЕРЕНИЙ

Теперь, после создания действующей, хотя и минималистичной игры, я предлагаю расширить наш кругозор. Мы можем представить в виде систем линейных уравнений весьма широкий спектр задач, а не только аркадные игры. Линейные уравнения, встречающиеся в повседневной практике, часто имеют больше двух неизвестных переменных. Такие уравнения описывают наборы точек в более чем двух измерениях. Нам трудно представить пространство с числом измерений больше трех, однако трехмерный случай может оказаться полезной моделью. Плоскости в трехмерном пространстве аналогичны прямым в двухмерном и тоже представлены линейными уравнениями.

7.3.1. Представление плоскостей в трех измерениях

Чтобы сделать более наглядными аналогии между прямыми и плоскостями, представим прямые в форме скалярных произведений. Как вы видели в упражнении 7.8 или, возможно, заметили сами, уравнение $ax + by = c$ представляет набор точек (x, y) на двухмерной плоскости, где скалярное произведение его и фиксированного вектора (a, b) равно фиксированному числу c. То есть уравнение $ax + by = c$ эквивалентно уравнению $(a, b) \cdot (x, y) = c$. Если вы не поняли, как интерпретировать это с геометрической точки зрения, рассмотрим этот вопрос.

Если в двухмерном пространстве есть точка и ненулевой вектор, то существует единственная прямая, перпендикулярная вектору и проходящая через эту точку (рис. 7.20).

Обозначив данную точку (x_0, y_0) и данный вектор (a, b), можно выразить критерий для точки (x, y), лежащей на перпендикулярной прямой. В частности, если (x, y) лежит на прямой, то вектор $(x - x_0, y - y_0)$ параллелен этой прямой и перпендикулярен (a, b), как показано на рис. 7.21.

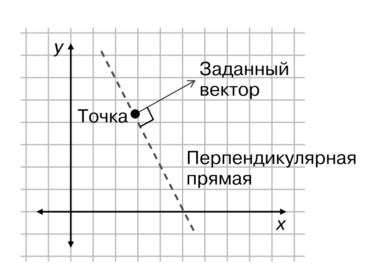


Рис. 7.20. Единственная прямая, проходящая через данную точку и перпендикулярная данному вектору

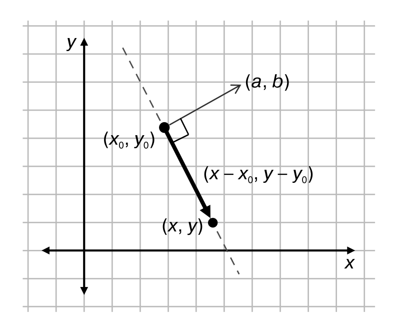


Рис. 7.21. Вектор $(x - x_0, y - y_0)$ параллелен прямой и, следовательно, перпендикулярен вектору (a, b)

Так как скалярное произведение двух перпендикулярных векторов равно нулю, это эквивалентно алгебраическому утверждению:

$$(a, b) \cdot (x - x_0, y - y_0) = 0.$$

Раскрыв скобки, получаем:

$$a(x - x_0) + b(y - y_0) = 0$$

или

$$ax + by = ax_0 + by_0.$$

Величина в правой части этого уравнения — константа, поэтому мы можем обозначить ее c, что даст общее уравнение прямой $ax + by = c$. Это удобная геометрическая интерпретация формулы $ax + by = c$, которую можно обобщить на три измерения.

Для данной точки и трехмерного вектора существует единственная *плоскость*, перпендикулярная вектору и проходящая через эту точку. Если обозначить вектор (a, b, c) и точку (x_0, y_0, z_0), то можно сказать, что если вектор (x, y, z) лежит в плоскости, то плоскость $(x - x_0, y - y_0, z - z_0)$ перпендикулярна плоскости (a, b, c). Эти рассуждения иллюстрирует рис. 7.22.

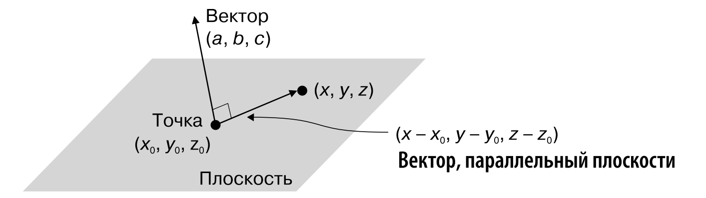


Рис. 7.22. Плоскость, параллельная вектору (a, b, c), проходит через точку (x_0, y_0, z_0)

Каждая точка на плоскости дает вектор, перпендикулярный к (a, b, c), и каждый вектор, перпендикулярный к (a, b, c), дает точку на плоскости. Эту перпендикулярность можно выразить как скалярное произведение двух векторов, поэтому уравнение, которому удовлетворяет каждая точка (x, y, z) на плоскости, имеет вид

$$(a, b, c) \cdot (x - x_0, y - y_0, z - z_0) = 0.$$

Раскрыв скобки, получаем:

$$ax + by + cz = ax_0 + by_0 + cz_0.$$

И поскольку правая часть уравнения — константа, мы можем заключить, что каждая плоскость в трехмерном пространстве определяется уравнением вида

$ax + by + cz = d$. Вычислительная задача в трехмерном случае состоит в том, чтобы решить, где пересекаются плоскости или какие значения (x, y, z) одновременно удовлетворяют нескольким линейным уравнениям, подобным этому.

7.3.2. Решение систем трех линейных уравнений

Пара непараллельных прямых на плоскости пересекается ровно в одной точке. Верно ли это утверждение для плоскостей? Нарисовав пару пересекающихся плоскостей, можно увидеть, что непараллельные плоскости пересекаются во множестве точек. На самом деле, как показано на рис. 7.23, на пересечении находится целая прямая, состоящая из бесконечного числа точек.

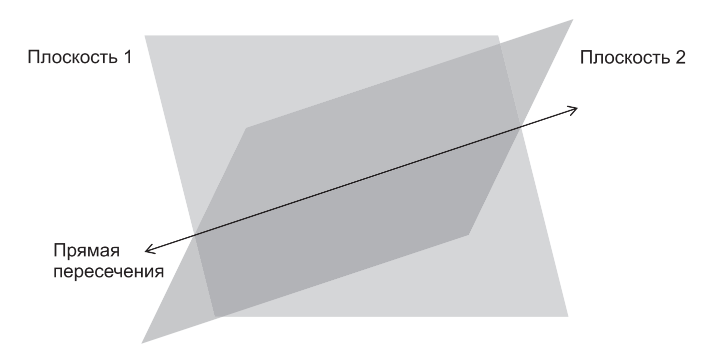


Рис. 7.23. Пересечение двух непараллельных плоскостей — это прямая

Если добавить третью плоскость, не параллельную этой прямой линии пересечения, то можно найти уникальную точку пересечения всех трех плоскостей. На рис. 7.24 показано, что каждая пара из трех плоскостей пересекается по прямой линии и эти линии пересечения имеют общую точку.

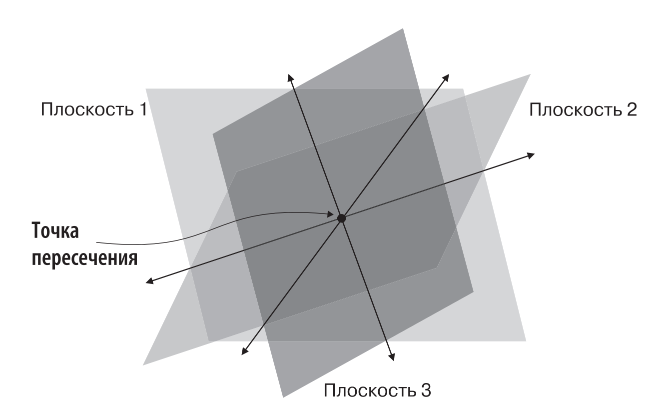


Рис. 7.24. Пересечение трех непараллельных плоскостей — точка

Чтобы определить координаты этой точки алгебраически, нужно решить систему из трех линейных уравнений с тремя переменными, каждое из которых представляет одну из плоскостей и имеет вид $ax + by + cz = d$. Система из трех линейных уравнений будет иметь вид

$$a_1x + b_1y + c_1z = d_1;$$
$$a_2x + b_2y + c_2z = d_2;$$
$$a_3x + b_3y + c_3z = d_3.$$

Каждая плоскость определяется четырьмя числами, a_i, b_i, c_i и d_i, где $i = 1$, 2 или 3 и является индексом рассматриваемой плоскости. Такие индексы удобно использовать, работая с системами линейных уравнений, где может быть много переменных, которым необходимо дать имена. Этих 12 чисел достаточно, чтобы найти точку (x, y, z) пересечения плоскостей, если она есть. Чтобы решить систему, можно преобразовать ее в матричное уравнение:

$$\begin{pmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \end{pmatrix}.$$

Рассмотрим практический пример. Пусть есть три плоскости, заданные следующими уравнениями:

$$x + y - z = -1;$$
$$2y - z = 3;$$
$$x + z = 2.$$

В примерах с исходным кодом для книги вы сможете увидеть, как построить эти плоскости с помощью Matplotlib. На рис. 7.25 показан результат.

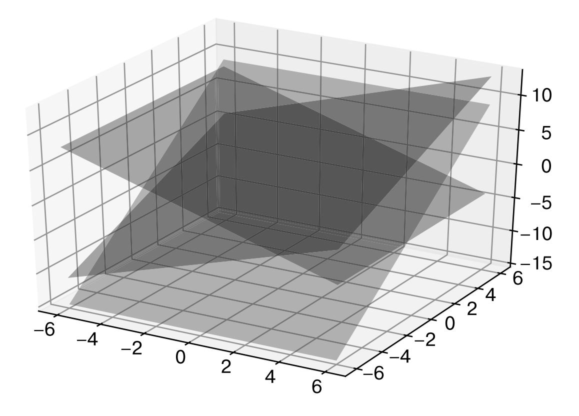


Рис. 7.25. Три плоскости, нарисованные с помощью Matplotlib

На рисунке плохо видно, но где-то там три плоскости пересекаются. Чтобы найти точку пересечения, нужно определить значения x, y и z, удовлетворяющие одновременно трем линейным уравнениям. И снова мы можем преобразовать систему в матричную форму и решить ее с помощью NumPy. Матричное уравнение, эквивалентное этой линейной системе, имеет вид

$$\begin{pmatrix} 1 & 1 & -1 \\ 0 & 2 & -1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -1 \\ 3 \\ 2 \end{pmatrix}.$$

Преобразовав матрицу и вектор в массивы NumPy, можно быстро найти векторное решение:

```
>>> matrix = np.array(((1,1,-1),(0,2,-1),(1,0,1)))
>>> vector = np.array((-1,3,2))
>>> np.linalg.solve(matrix,vector)
array([-1., 3., 3.])
```

Результат говорит нам, что $(-1, 3, 3)$ — это точка (x, y, z) пересечения всех трех плоскостей, которая одновременно удовлетворяет всем трем линейным уравнениям.

Мы легко вычислили этот результат с помощью NumPy, но как вы могли видеть, представить визуально систему из трех линейных уравнений оказалось немного сложнее. Вообразить системы линейных уравнений с еще большим числом измерений еще сложнее, если вообще возможно, но механически они решаются точно так же. Аналогия с линией или плоскостью в любом количестве измерений называется *гиперплоскостью*, и задача сводится к определению точек пересечения нескольких гиперплоскостей.

7.3.3. Алгебраическое изучение гиперплоскостей

Если сказать точнее, гиперплоскость в n измерениях — это решение линейного уравнения с n неизвестными переменными. Прямая — это одномерная гиперплоскость в двухмерном пространстве, а плоскость — двухмерная гиперплоскость в трехмерном пространстве. Как нетрудно догадаться, стандартная форма линейного уравнения с четырьмя неизвестными переменными имеет вид

$$aw + bx + cy + dz = e.$$

Множество решений (w, x, y, z) образует область — трехмерную гиперплоскость в четырехмерном пространстве. Мы должны быть осторожными, используя прилагательное «трехмерный», потому что это не обязательно трехмерное векторное подпространство в $\mathbb{R}^4$. Здесь можно провести аналогию с двухмерным случаем: одни прямые, проходящие через начало координат в двухмерном пространстве, являются векторными подпространствами $\mathbb{R}^2$, а другие — нет. Трехмерная

гиперплоскость независимо от того, является она векторным пространством или нет, трехмерна в том смысле, что имеет три линейно независимых направления, по которым можно двигаться в множестве решений, как есть два линейно независимых направления, по которым можно двигаться на любой плоскости. В конце этого раздела я поместил мини-проект, который поможет вам проверить свое понимание этой концепции.

Записывая линейные уравнения с еще большим числом измерений, есть риск исчерпать буквы для обозначения координат и коэффициентов. Чтобы избежать этой проблемы, будем использовать буквы с нижними индексами. Например, линейное уравнение с четырьмя переменными можно записать в стандартной форме так:

$$a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4 = b.$$

Коэффициентами здесь являются a_1, a_2, a_3 и a_4, а четырехмерный вектор определяется координатами (x_1, x_2, x_3, x_4). С таким же успехом мы могли бы записать линейное уравнение с 10 переменными:

$$a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4 + a_5x_5 + a_6x_6 + a_7x_7 + a_8x_8 + a_9x_9 + a_{10}x_{10} = b.$$

Когда ясно видна закономерность следования слагаемых в уравнении, для экономии места иногда используется многоточие (...). Например, предыдущее уравнение можно записать так: $a_1x_1 + a_2x_2 + ... + a_{10}x_{10} = b$. Еще одно компактное обозначение, которое вы увидите далее, включает символ суммирования Σ — греческую букву *«сигма»*. Например, равенство некоторому числу b суммы членов в форме a_ix_i с индексом i в диапазоне от $i = 1$ до $i = 10$ можно кратко записать, используя математическую нотацию:

$$\sum_{i=1}^{10} a_ix_i = b.$$

Это уравнение означает то же самое, что и предыдущее, просто имеет более краткую запись. С каким бы числом измерений n мы ни работали, стандартная форма линейного уравнения имеет один и тот же вид:

$$a_1x_1 + a_2x_2 + ... + a_nx_n = b.$$

Чтобы представить систему m линейных уравнений в n измерениях, нужны дополнительные индексы. Массив констант слева от знака равенства можно обозначить как a_{ij}, где индекс i обозначает номер уравнения, а индекс j — координату (x_j), которая умножается на константу, например:

$$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1;$$
$$a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2;$$
$$...$$
$$a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n = b_m.$$

Как видите, здесь я использовал многоточие, чтобы пропустить уравнения с третьего по $(m - 1)$-е. Всего имеется m уравнений и n констант в каждом из них, поэтому всего мы имеем mn констант вида a_{ij}. В правой части m констант, по одной на уравнение: $b_1, b_2, ..., b_m$.

Независимо от количества измерений (то есть количества неизвестных переменных) и количества уравнений такую систему можно представить в матричном виде. Так, предыдущую систему с n неизвестными и m уравнениями можно переписать, как показано на рис. 7.26.

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

Рис. 7.26. Система из m линейных уравнений с n неизвестными, записанная в матричной форме

7.3.4. Подсчет числа измерений, уравнений и решений

Мы видели, что системы двух и трех линейных уравнений могут не иметь решения или иметь множество решений. Как узнать, имеет ли система из m уравнений с n неизвестными одно уникальное решение? Иначе говоря, как определить, имеют ли m гиперплоскостей в n измерениях единственную точку пересечения? Мы подробно обсудим этот вопрос в последнем разделе главы, но уже сейчас можем сделать один важный вывод.

В двухмерном пространстве пара прямых может пересекаться в одной точке. Не всегда (например, если прямые параллельны), но может. Эквивалентное утверждение на языке алгебры: система двух линейных уравнений с двумя переменными может иметь единственное решение.

В трехмерном пространстве три плоскости могут пересекаться в одной точке. Это верно не для всех плоскостей, но три — это минимальное количество плоскостей (или линейных уравнений), необходимых для задания точки в трех измерениях. При двух плоскостях мы можем получить как минимум одномерное пространство возможных решений — линию пересечения. Алгебраически это означает, что для получения уникального решения в двухмерном пространстве нужны два линейных уравнения, а в трехмерном пространстве — три. В общем случае нужно иметь n линейных уравнений, чтобы получить уникальное решение в n измерениях.

Вот пример с использованием четырехмерных координат (x_1, x_2, x_3, x_4), который может показаться слишком простым, но он полезен благодаря своей конкретности. Пусть первым линейным уравнением будет $x_4 = 0$. Решения этого линейного уравнения образуют трехмерную гиперплоскость, состоящую из векторов вида $(x_1, x_2, x_3, 0)$. Очевидно, что это трехмерное пространство решений, одновременно являющееся векторным подпространством в $\mathbb{R}^4$ с базисом $(1, 0, 0, 0)$, $(0, 1, 0, 0)$, $(0, 0, 1, 0)$.

Пусть второе линейное уравнение имеет вид $x_2 = 0$. Его решения также образуют трехмерную гиперплоскость. Пересечение этих двух трехмерных

гиперплоскостей — двухмерное пространство, состоящее из векторов вида $(x_1, 0, x_3, 0)$, удовлетворяющих обоим уравнениям. Если бы мы могли изобразить его, то увидели бы двухмерную плоскость в четырехмерном пространстве — плоскость, натянутую на (1, 0, 0, 0) и (0, 0, 1, 0).

Добавив еще одно линейное уравнение, $x_1 = 0$, определяющее свою гиперплоскость, получаем одномерное пространство решений всех трех уравнений. Векторы в этом пространстве лежат на прямой линии в четырехмерном пространстве и имеют форму $(0, 0, x_3, 0)$. Эта прямая совпадает с осью x_3, которая является одномерным подпространством в $\mathbb{R}^4$.

Наконец, наложив четвертое линейное уравнение, $x_3 = 0$, получаем единственное возможное решение (0, 0, 0, 0) — нульмерное векторное пространство. Утверждения $x_4 = 0$, $x_2 = 0$, $x_1 = 0$ и $x_3 = 0$ на самом деле являются линейными уравнениями, но они настолько просты, что их решение очевидно: $(x_1, x_2, x_3, x_4) = (0, 0, 0, 0)$. Каждый раз, добавляя уравнение, мы уменьшаем размерность пространства решений на единицу, и так до тех пор, пока не получим нульмерное пространство, состоящее из одной точки (0, 0, 0, 0).

Если бы мы выбрали другие уравнения, то каждый шаг не был бы таким же ясным. Нам пришлось бы проверить, действительно ли каждая последующая гиперплоскость уменьшает размерность пространства решений на единицу. Например, начав с

$$x_1 = 0$$

и

$$x_2 = 0,$$

мы бы сузили множество решений до двухмерного пространства, но последующее добавление еще одного уравнения

$$x_1 + x_2 = 0$$

не влияет на пространство решений. Поскольку x_1 и x_2 уже ограничены равенством нулю, уравнение $x_1 + x_2 = 0$ выполняется автоматически. Поэтому третье уравнение не добавляет ничего нового в множество решений.

В первом случае четыре измерения с тремя линейными уравнениями оставили нам $(4 - 3 = 1)$-мерное пространство решений. Но во втором случае три уравнения описывают менее конкретное двухмерное пространство решений. n линейных уравнений в n измерениях (с n неизвестными переменными) могут иметь единственное решение — нульмерное пространство решений, но так бывает не всегда. В общем случае, когда идет работа с n измерениями, самая малая размерность пространства решений, которое можно получить с помощью m линейных уравнений, равна $n - m$. В этом случае мы называем систему линейных уравнений *независимой*.

Каждый базисный вектор дает новое независимое направление, в котором можно двигаться в пространстве. Независимые направления в пространстве иногда

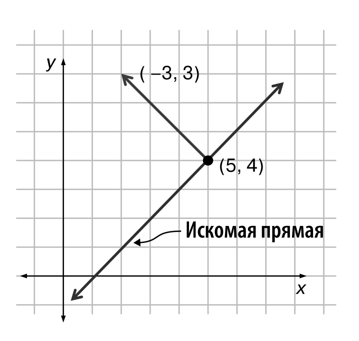
называют *степенями свободы*: направление z, например, «освободило» нас от плоскости и перенесло в более обширное трехмерное пространство. Напротив, каждое независимое линейное уравнение, которое вводится в систему, накладывает ограничение — уничтожает степень свободы и ограничивает пространство решений меньшим числом измерений. Когда количество независимых степеней свободы (размерностей) равно количеству независимых ограничений (линейных уравнений), степеней свободы не остается и мы получаем единственную точку.

Это важный философский момент в линейной алгебре, который вы сможете изучить подробнее, выполняя следующие мини-проекты. В последнем разделе этой главы мы свяжем понятия независимых уравнений и линейно независимых векторов.

7.3.5. Упражнения

Упражнение 7.16. Определите уравнение прямой, проходящей через точку (5, 4) и перпендикулярной вектору (–3, 3).

Решение. Вот иллюстрация к этому упражнению.



Для каждой точки (x, y) на прямой вектор $(x - 5, y - 4)$ параллелен этой прямой и, следовательно, перпендикулярен вектору (–3, 3). Это означает, что скалярное произведение $(x - 5, y - 4) \cdot (-3, 3)$ равно нулю для любой точки (x, y) на прямой. Раскрыв скобки, получаем уравнение $-3x + 15 + 3y - 12 = 0$, которое после сокращения подобных членов и реорганизации принимает вид $-3x + 3y = -3$. Мы можем разделить обе стороны на –3, чтобы получить более простое эквивалентное уравнение $x - y = 1$.

Упражнение 7.17. Мини-проект. Взгляните на систему двух линейных уравнений с четырьмя переменными:

$$x_1 + 2x_2 + 2x_3 + x_4 = 0;$$
$$x_1 - x_4 = 0.$$

Объясните алгебраически (не геометрически), почему решения образуют векторное подпространство в четырехмерном пространстве.

Решение. Мы можем показать, что если (a_1, a_2, a_3, a_4) и (b_1, b_2, b_3, b_4) — два решения, то их линейная комбинация также является решением. Это означало бы, что набор решений содержит все линейные комбинации его векторов, и это делает его векторным подпространством.

Начнем с предположения, что (a_1, a_2, a_3, a_4) и (b_1, b_2, b_3, b_4) являются решениями обоих линейных уравнений, а это явно означает:

$$a_1 + 2a_2 + 2a_3 + a_4 = 0;$$
$$b_1 + 2b_2 + 2b_3 + b_4 = 0;$$
$$a_1 - a_4 = 0;$$
$$b_1 - b_4 = 0.$$

С выбранными скалярами c и d линейная комбинация $c(a_1, a_2, a_3, a_4) + d(b_1, b_2, b_3, b_4)$ равна $(ca_1 + db_1, ca_2 + db_2, ca_3 + db_3, ca_4 + db_4)$. Является ли это решением двух уравнений? Это можно узнать, подставив четыре координаты, x_1, x_2, x_3 и x_4. Первое уравнение

$$x_1 + 2x_2 + 2x_3 + x_4$$

превращается в

$$(ca_1 + db_1) + 2(ca_2 + db_2) + 2(ca_3 + db_3) + (ca_4 + db_4).$$

Раскрываем скобки:

$$ca_1 + db_1 + 2ca_2 + 2db_2 + 2ca_3 + 2db_3 + ca_4 + db_4.$$

Переупорядочиваем:

$$c(a_1 + 2a_2 + 2a_3 + a_4) + d(b_1 + 2b_2 + 2b_3 + b_4).$$

Поскольку $a_1 + 2a_2 + 2a_3 + a_4$ и $b_1 + 2b_2 + 2b_3 + b_4$ равны нулю, то и все выражение равно нулю:

$$c(a_1 + 2a_2 + 2a_3 + a_4) + d(b_1 + 2b_2 + 2b_3 + b_4) = c \cdot 0 + d \cdot 0 = 0.$$

Это означает, что линейная комбинация является решением первого уравнения. Точно так же, подставив линейную комбинацию во второе уравнение, мы увидим, что она оказывается решением и этого уравнения:

$$(ca_1 + db_1) - (ca_4 + db_4) = c(a_1 - a_4) + d(b_1 - b_4) = c \cdot 0 + d \cdot 0 = 0.$$

Любая линейная комбинация любых двух решений также является решением, поэтому множество решений содержит все ее линейные комбинации. А это означает, что набор решений — это векторное подпространство в четырехмерном пространстве.

Упражнение 7.18. Напишите стандартное уравнение плоскости, проходящей через точку (1, 1, 1) и перпендикулярной вектору (1, 1, 1).

Решение. Для любой точки (x, y, z) на плоскости вектор $(x - 1, y - 1, z - 1)$ перпендикулярен (1, 1, 1). Это означает, что скалярное произведение $(x - 1, y - 1, z - 1) \cdot (1, 1, 1)$ равно нулю для любых значений x, y и z, соответствующих точке на плоскости. В результате получаем $(x - 1) + (y - 1) + (z - 1) = 0$ или $x + y + z = 3$ — уравнение плоскости в стандартной форме.

Упражнение 7.19. Мини-проект. Напишите на Python функцию, которая принимает три трехмерные точки и возвращает уравнение плоскости в стандартной форме, на которой они лежат. Например, учитывая, что стандартное уравнение имеет вид $ax + by + cz = d$, функция должна вернуть кортеж (a, b, c, d).

Подсказка. Разности любых пар из трех векторов параллельны плоскости, поэтому векторные произведения разностей будут перпендикулярны.

Решение. Если заданы точки p_1, p_2 и p_3, то разности векторов, такие как $p_3 - p_1$ и $p_2 - p_1$, параллельны плоскости. Тогда векторное произведение $(p_2 - p_1) \times (p_3 - p_1)$ будет перпендикулярно плоскости. (Все это верно, если точки p_1, p_2 и p_3 образуют треугольник и поэтому разности не параллельны.) Имея точку на плоскости (например, p_1) и перпендикулярный вектор, можно повторить процесс определения стандартного вида решения, как в двух предыдущих упражнениях:

```
from vectors import *

def plane_equation(p1,p2,p3):
    parallel1 = subtract(p2,p1)
    parallel2 = subtract(p3,p1)
    a,b,c = cross(parallel1, parallel2)
    d = dot((a,b,c), p1)
    return a,b,c,d
```

Например, для трех точек на плоскости $x + y + z = 3$ из предыдущего упражнения

```
>>> plane_equation((1,1,1), (3,0,0), (0,3,0))
(3, 3, 3, 9)
```

В результате получился кортеж (3, 3, 3, 9), означающий $3x + 3y + 3z = 9$, что эквивалентно $x + y + z = 3$. То есть написанная нами функция работает правильно!

Упражнение 7.20. Сколько констант a_{ij} содержится в следующем матричном уравнении? Сколько всего уравнений? Сколько неизвестных? Запишите полное матричное уравнение и полную систему линейных уравнений — все без многоточий.

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{17} \\ a_{21} & a_{22} & \cdots & a_{27} \\ \vdots & \vdots & \ddots & \vdots \\ a_{51} & a_{52} & \cdots & a_{57} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_7 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_5 \end{pmatrix}$$

Система линейных уравнений в матричной форме и в сокращенной записи

Решение. Для ясности запишем сначала полное уравнение в матричной форме.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} & a_{57} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{pmatrix}$$

Полная версия уравнения в матричной форме

Всего в этой матрице $5 \cdot 7 = 35$ элементов и 35 констант a_{ij} в левой части системы линейных уравнений. Имеется семь неизвестных переменных, $x_1, x_2, ..., x_7$, и пять уравнений (по одному на строку матрицы). Получить полную линейную систему можно, выполнив умножение матриц:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 + a_{15}x_5 + a_{16}x_6 + a_{17}x_7 = b_1;$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 + a_{25}x_5 + a_{26}x_6 + a_{27}x_7 = b_2;$$
$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 + a_{35}x_5 + a_{36}x_6 + a_{37}x_7 = b_3;$$
$$a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 + a_{45}x_5 + a_{46}x_6 + a_{47}x_7 = b_4;$$
$$a_{51}x_1 + a_{52}x_2 + a_{53}x_3 + a_{54}x_4 + a_{55}x_5 + a_{56}x_6 + a_{57}x_7 = b_5.$$

Полная система линейных уравнений, соответствующая матричному уравнению

Теперь четко видно, сколько утомительной работы позволяет избежать краткая форма записи!

Упражнение 7.21. Напишите следующее линейное уравнение без сокращения суммирования:

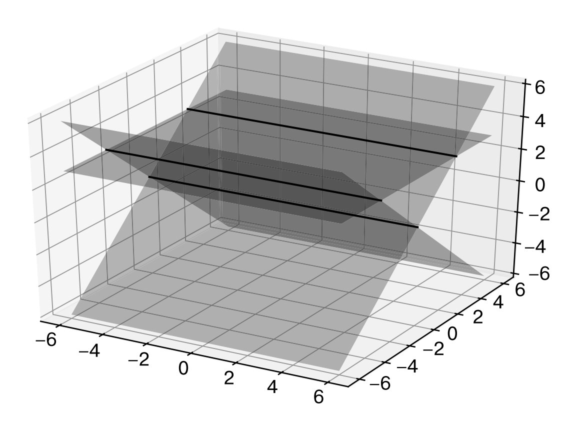
$$\sum_{i=1}^{3} x_i = 1.$$

Как выглядит набор решений геометрически?

Решение. Левая сторона уравнения — это сумма членов вида x_i для i в диапазоне от 1 до 3: $x_1 + x_2 + x_3 = 1$. Это стандартная форма линейного уравнения с тремя переменными, поэтому его решения образуют плоскость в трехмерном пространстве.

Упражнение 7.22. Нарисуйте три непараллельные друг другу плоскости, которые не имеют общей точки пересечения. (А еще лучше найдите их уравнения и нарисуйте их программно!)

Решение. Вот три такие плоскости: $z + y = 0$, $z - y = 0$, $z = 3$, и их изображение.



Три непараллельные плоскости, не имеющие общей точки пересечения

Я нарисовал линии пересечения трех пар плоскостей. Эти линии параллельны друг другу. Поскольку они нигде не пересекаются, то и плоскости не имеют общей точки пересечения. Это похоже на пример, показанный в главе 6: три вектора могут быть линейно зависимыми, даже если среди них нет параллельных пар.

Упражнение 7.23. Пусть есть m линейных уравнений и n неизвестных переменных. Что говорят следующие значения m и n о существовании единственного решения?

1. $m = 2, n = 2$.
2. $m = 2, n = 7$.
3. $m = 5, n = 5$.
4. $m = 3, n = 2$.

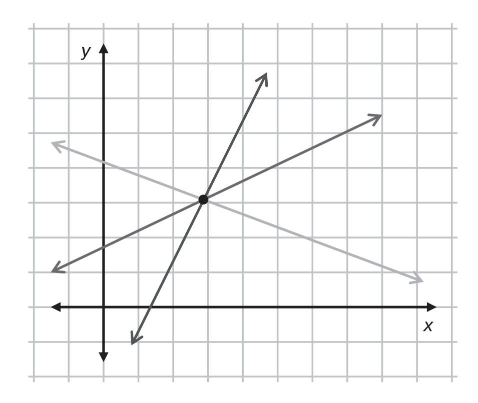
Решение

Система с двумя линейными уравнениями и двумя неизвестными может иметь единственное решение. Два уравнения представляют линии на плоскости, которые будут пересекаться в единственной точке, если они не параллельны.

Система с двумя линейными уравнениями и семью неизвестными не может иметь единственного решения. Даже если предположить, что шестимерные гиперплоскости, определяемые этими уравнениями, не параллельны, два уравнения дают пятимерное пространство решений.

Система с пятью линейными уравнениями и пятью неизвестными может иметь единственное решение, если уравнения независимы.

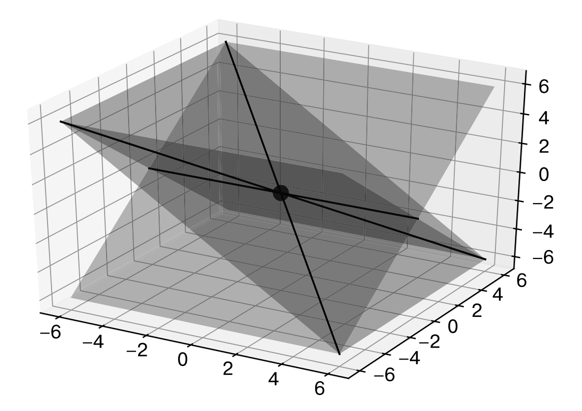
Система с тремя линейными уравнениями и двумя неизвестными может иметь единственное решение, но для этого требуется удача. Наличие решения означает, что третья линия проходит через точку пересечения первых двух линий, что маловероятно, но возможно.



Три прямые на плоскости, пересекающиеся в одной точке

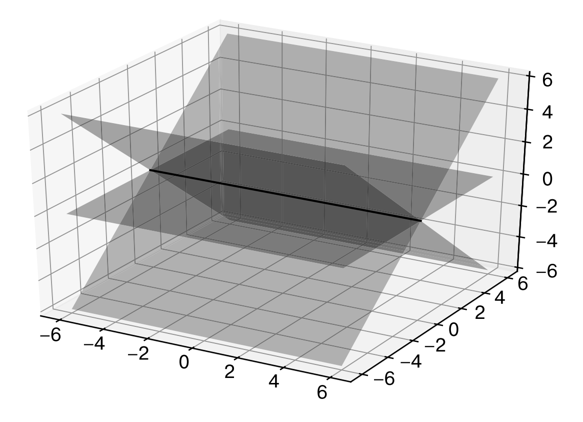
Упражнение 7.24. Найдите три плоскости, имеющие единственную общую точку пересечения, три плоскости, пересечение которых — это прямая, и три плоскости, пересечением которых является плоскость.

Решение. Плоскости $z - y = 0$, $z + y = 0$ и $z + x = 0$ пересекаются в одной точке $(0, 0, 0)$. Большинство случайно выбранных плоскостей пересекаются в одной точке.



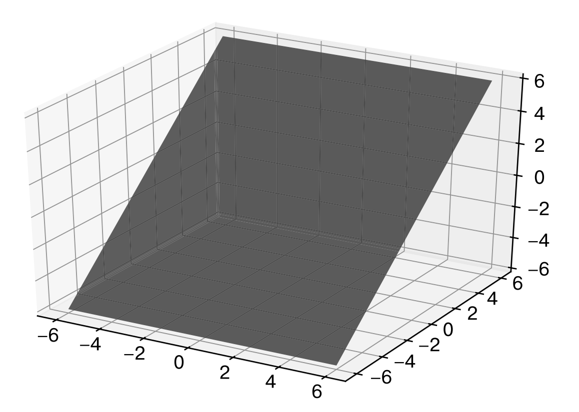
Три плоскости, пересекающиеся в одной точке

Пересечение плоскостей $z - y = 0$, $z + y = 0$ и $z = 0$ — это прямая, а именно ось x. Обратите внимание на то, что y и z ограничены нулем, а x вообще отсутствует в уравнениях, поэтому на величину x нет ограничений. Соответственно, решением является любой вектор $(x, 0, 0)$, лежащий на оси x.



Три плоскости, точки пересечения которых образуют прямую

Наконец, если все три уравнения представляют одну и ту же плоскость, то вся эта плоскость — это набор решений. Например, уравнения $z - y = 0$, $2z - 2y = 0$ и $3z - 3y = 0$ представляют одну и ту же плоскость.



Три одинаковые плоскости наложены друг на друга, в этом случае множество решений — это вся плоскость

Упражнение 7.25. Найдите решение системы линейных уравнений в пятимерном пространстве без использования Python: $x_5 = 3$, $x_2 = 1$, $x_4 = -1$, $x_1 = 0$ и $x_1 + x_2 + x_3 = -2$. Подтвердите ответ с помощью NumPy.

Решение. Четыре уравнения из пяти задают значения координат, отсюда следует, что решение имеет вид $(0, 1, x_3, -1, 3)$. Осталось только выполнить некоторые вычисления с последним уравнением, чтобы узнать значение x_3. Подставив значения в уравнение $x_1 + x_2 + x_3 = -2$, получаем $0 + 1 + x_3 = -2$, откуда следует, что $x_3 = -3$. Соответственно, точкой решения является $(0, 1, -3, -1, 3)$. Преобразовав эту систему в матричную форму, ее можно решить с помощью NumPy, чтобы подтвердить наш результат:

```
>>> matrix =
np.array(((0,0,0,0,1),(0,1,0,0,0),(0,0,0,1,0),(1,0,0,0,0),(1,1,1,0,0)))
>>> vector = np.array((3,1,-1,0,-2))
>>> np.linalg.solve(matrix,vector)
array([ 0., 1., -3., -1., 3.])
```

Упражнение 7.26. Мини-проект. В пространстве любой мерности существует единичная матрица, которая действует как идентичное отображение. То есть при умножении n-мерной единичной матрицы I на любой вектор $\mathbf{v}$ получается тот же самый вектор $\mathbf{v}$, следовательно, $I\mathbf{v} = \mathbf{v}$.

Это означает, что $I\mathbf{v} = \mathbf{w}$ — простая для решения система линейных уравнений: один из возможных ответов для $\mathbf{v}$ — это $\mathbf{v} = \mathbf{w}$. Суть этого мини-проекта состоит в том, что можно начать с системы линейных уравнений $A\mathbf{v} = \mathbf{w}$ и умножить обе части на другую матрицу, B, такую, что $(BA) = I$. Если это условие выполняется, то выполняются условия $(BA)\mathbf{v} = B\mathbf{w}$ и $I\mathbf{v} = B\mathbf{w}$ или $\mathbf{v} = B\mathbf{w}$. Иначе говоря, если есть система $A\mathbf{v} = \mathbf{w}$ и подходящая матрица B, то $B\mathbf{w}$ — решение этой системы. Такая матрица B называется *обратной* матрицей для A.

Еще раз рассмотрим систему уравнений, которую мы решили в разделе 7.3.2:

$$\begin{pmatrix} 1 & 1 & -1 \\ 0 & 2 & -1 \\ 1 & 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -1 \\ 3 \\ 2 \end{pmatrix}.$$

Примените функцию `numpy.linalg.inv(matrix)` из библиотеки NumPy, которая находит и возвращает матрицу, обратную заданной, чтобы найти обратную матрицу для матрицы в левой части уравнения. Затем умножьте обе части на эту матрицу, чтобы найти решение линейной системы. Сравните свои результаты с результатами, которые мы получили с помощью той же библиотеки NumPy.

Совет. Можете использовать умножение матриц `numpy.matmul` из библиотеки NumPy, чтобы упростить вычисления.

Решение. Сначала найдем обратную матрицу с помощью NumPy:

```
>>> matrix = np.array(((1,1,-1),(0,2,-1),(1,0,1)))
>>> vector = np.array((-1,3,2))
>>> inverse = np.linalg.inv(matrix)
>>> inverse
array([[ 0.66666667, -0.33333333, 0.33333333],
       [-0.33333333, 0.66666667, 0.33333333],
       [-0.66666667, 0.33333333, 0.66666667]])
```

Умножение обратной матрицы на исходную дает единичную матрицу с единицами по диагонали и нулями в других элементах, хотя и с некоторой ошибкой, обусловленной конечной точностью вычислений с плавающей точкой:

```
>>> np.matmul(inverse,matrix)
array([[ 1.00000000e+00, 1.11022302e-16, -1.11022302e-16],
       [ 0.00000000e+00, 1.00000000e+00, 0.00000000e+00],
       [ 0.00000000e+00, 0.00000000e+00, 1.00000000e+00]])
```

Хитрость заключается в том, чтобы умножить обе части матричного уравнения на эту обратную матрицу. Здесь я округлил значения в обратной матрице для удобочитаемости. Мы уже знаем, что первые два сомножителя слева — это исходная и обратная матрицы, поэтому можем выполнить упрощение:

$$\begin{pmatrix} 0{,}667 & -0{,}333 & 0{,}333 \\ -0{,}333 & 0{,}667 & 0{,}333 \\ -0{,}667 & 0{,}333 & 0{,}667 \end{pmatrix}\begin{pmatrix} 1 & -1 & 0 \\ 0 & -1 & -1 \\ 1 & 0 & 2 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0{,}667 & -0{,}333 & 0{,}333 \\ -0{,}333 & 0{,}667 & 0{,}333 \\ -0{,}667 & 0{,}333 & 0{,}667 \end{pmatrix}\begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix};$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0{,}667 & -0{,}333 & 0{,}333 \\ -0{,}333 & 0{,}667 & 0{,}333 \\ -0{,}667 & 0{,}333 & 0{,}667 \end{pmatrix}\begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix};$$

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0{,}667 & -0{,}333 & 0{,}333 \\ -0{,}333 & 0{,}667 & 0{,}333 \\ -0{,}667 & 0{,}333 & 0{,}667 \end{pmatrix}\begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix}.$$

Умножение обеих сторон системы на обратную матрицу и упрощение

Это дает нам явную формулу для решения системы уравнений относительно (x, y, z), нам нужно лишь перемножить матрицы. Оказывается, `numpy.matmul` также может умножать матрицы на векторы:

```
>>> np.matmul(inverse, vector)
array([-1., 3., 3.])
```

Это тот же результат, который мы получили ранее в этой главе.

7.4. ИЗМЕНЕНИЕ БАЗИСА ПУТЕМ РЕШЕНИЯ ЛИНЕЙНЫХ УРАВНЕНИЙ

Понятие линейной независимости векторов тесно связано с понятием независимости линейных уравнений. Эта связь обусловлена тем, что решение системы линейных уравнений эквивалентно переписыванию векторов в координатах другого базиса. Посмотрим, что это означает, на примере двухмерного пространства.

Записывая координаты вектора, такие как (4, 3), мы неявно записываем вектор как линейную комбинацию векторов стандартного базиса:

$$(4, 3) = 4\mathbf{e}_1 + 3\mathbf{e}_2.$$

В предыдущей главе вы узнали, что стандартный базис, состоящий из векторов $\mathbf{e}_1 = (1, 0)$ и $\mathbf{e}_2 = (0, 1)$, не единственный. Например, пара векторов $\mathbf{u}_1 = (1, 1)$ и $\mathbf{u}_2 = (-1, 1)$ тоже образует базис для $\mathbb{R}^2$. Любой двухмерный вектор можно записать в виде линейной комбинации $\mathbf{e}_1$ и $\mathbf{e}_2$, и точно так же любой двухмерный вектор можно записать в виде линейной комбинации $\mathbf{u}_1$ и $\mathbf{u}_2$. Можно подобрать такие значения c и d, для которых уравнение

$$c \cdot (1, 1) + d \cdot (-1, 1) = (4, 2)$$

будет верным, но эти значения неочевидны. На рис. 7.27 показано визуальное представление этой задачи.

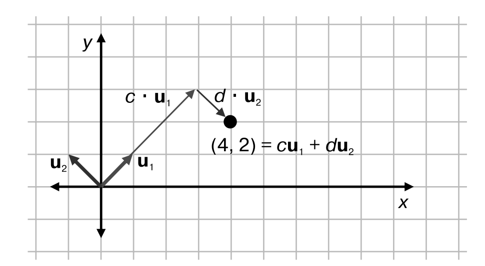


Рис. 7.27. Представление вектора (4, 2) в виде линейной комбинации $\mathbf{u}_1 = (1, 1)$ и $\mathbf{u}_2 = (-1, 1)$

Как линейная комбинация это уравнение эквивалентно матричному уравнению

$$\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \end{pmatrix}.$$

Это тоже система линейных уравнений! В этом случае неизвестный вектор записывается как (c, d), а не как (x, y), а линейные уравнения, скрытые в матричном уравнении, имеют вид $c - d = 4$ и $c + d = 2$. Имеется двухмерное пространство векторов (c, d), определяемое различными линейными комбинациями $\mathbf{u}_1$ и $\mathbf{u}_2$, но только один удовлетворяет этим двум уравнениям.

Любая выбранная пара (c, d) определяет свою линейную комбинацию. В качестве примера рассмотрим произвольное значение (c, d), скажем, $(c, d) = (3, 1)$. Вектор (3, 1) не принадлежит тому же векторному пространству, что и $\mathbf{u}_1$ и $\mathbf{u}_2$, — он находится в векторном пространстве пар (c, d), каждая из которых описывает различные линейные комбинации $\mathbf{u}_1$ и $\mathbf{u}_2$. Точка $(c, d) = (3, 1)$ описывает определенную линейную комбинацию в исходном двухмерном пространстве: $3\mathbf{u}_1 + 1\mathbf{u}_2$ приводит нас к точке $(x, y) = (2, 4)$, как показано на рис. 7.28.

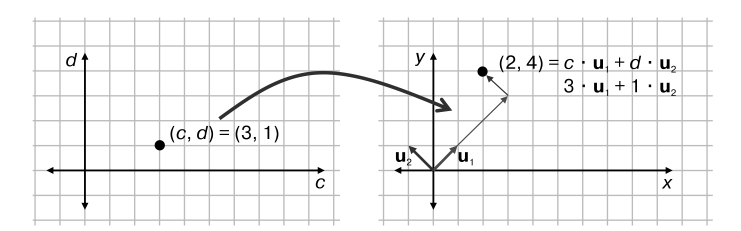


Рис. 7.28. Существует двухмерное пространство значений (*c*, *d*), где (*c*, *d*) = (3, 1) и дает линейную комбинацию $3\mathbf{u}_1 + 1\mathbf{u}_2 = (2, 4)$

Напомню, что мы пытаемся представить (4, 2) как линейную комбинацию $\mathbf{u}_1$ и $\mathbf{u}_2$, то есть это не та линейная комбинация, которую мы искали. Чтобы в результате линейной комбинации $c\mathbf{u}_1 + d\mathbf{u}_2$ получился вектор (4, 2), должны выполняться условия $c - d = 4$ и $c + d = 2$, как было показано ранее.

Нарисуем систему линейных уравнений в плоскости *cd*. Взглянув на рисунок, можно сказать, что условиям $c + d = 2$ и $c - d = 4$ удовлетворяет точка (3, –1). Это дает нам пару скаляров, которые можно использовать в линейной комбинации, чтобы получить (4, 2) из $\mathbf{u}_1$ и $\mathbf{u}_2$, как показано на рис. 7.29.

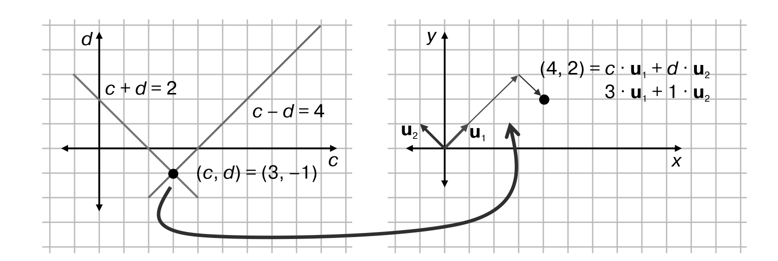


Рис. 7.29. Точка (*c*, *d*) = (3, –1) удовлетворяет обоим условиям, $c + d = 2$ и $c - d = 4$. Следовательно, она описывает искомую линейную комбинацию

Теперь мы можем записать (4, 2) как линейную комбинацию двух различных пар базисных векторов: $(4, 2) = 4\mathbf{e}_1 + 2\mathbf{e}_2$ и $(4, 2) = 3\mathbf{u}_1 - 1\mathbf{u}_2$. Напомню, что координаты (4, 2) — это скаляры линейной комбинации $4\mathbf{e}_1 + 2\mathbf{e}_2$. Если начертить оси иначе, то $\mathbf{u}_1$ и $\mathbf{u}_2$ вполне могли бы стать стандартным базисом, тогда вектор выражался бы как $3\mathbf{u}_1 - \mathbf{u}_2$ и мы говорили бы, что он имеет координаты (3, 1). Чтобы подчеркнуть, что координаты определяются выбором базиса, можно сказать, что вектор имеет координаты (4, 2) относительно стандартного базиса и координаты (3, –1) относительно базиса, состоящего из $\mathbf{u}_1$ и $\mathbf{u}_2$.

Определение координат вектора относительно другого базиса — это пример задачи, которая фактически является системой линейных уравнений. Это важный пример, потому что так можно представить любую систему линейных уравнений.

Для полноты картины рассмотрим еще один пример, на этот раз в трехмерном пространстве.

7.4.1. Решение трехмерного примера

Начнем с примера системы линейных уравнений трех переменных, а затем обсудим ее интерпретацию. Вместо матрицы 2×2 и двухмерного вектора будем использовать матрицу 3×3 и трехмерный вектор:

$$\begin{pmatrix} 1 & -1 & 0 \\ 0 & -1 & -1 \\ 1 & 0 & 2 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ -7 \end{pmatrix}.$$

Неизвестное здесь — трехмерный вектор; чтобы идентифицировать его, нужно найти три числа. Умножение матриц можно разбить на три уравнения:

$$1 \cdot x - 1 \cdot y + 0 \cdot z = 1;$$

$$0 \cdot x - 1 \cdot y - 1 \cdot z = 3;$$

$$1 \cdot x + 0 \cdot y + 2 \cdot z = -7.$$

Это система трех линейных уравнений с тремя неизвестными, которая в стандартной форме записи выглядит как $ax + by + cz = d$. В следующем разделе мы рассмотрим геометрическую интерпретацию трехмерных линейных уравнений. (Как оказывается, они представляют плоскости в трехмерном пространстве, а не линии, как в двухмерном пространстве.)

А пока посмотрим на эту систему как на линейную комбинацию, коэффициенты которой нужно определить. Предыдущее матричное уравнение эквивалентно следующему:

$$x\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} + y\begin{pmatrix} -1 \\ -1 \\ 0 \end{pmatrix} + z\begin{pmatrix} 0 \\ -1 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ -7 \end{pmatrix}.$$

Чтобы решить это уравнение, нужно задать вопрос: какая линейная комбинация векторов (1, 0, 1), (−1, −1, 0) и (0, −1, 2) дает вектор (1, 3, – 7)? Эту систему труднее изобразить, чем двухмерный пример, и труднее вычислить ответ вручную. Но, к счастью, NumPy может решать системы линейных уравнений с тремя неизвестными, поэтому мы просто передадим библиотеке матрицу 3×3 с трехмерным вектором:

```
>>> import numpy as np
>>> w = np.array((1,3,-7))
>>> a = np.array(((1,-1,0),(0,-1,-1),(1,0,2)))
>>> np.linalg.solve(a,w)
array([ 3., 2., -5.])
```

Значения, являющиеся решением нашей системы линейных уравнений, — $x = 3$, $y = 2$ и $z = -5$. Иначе говоря, это искомые коэффициенты в заданной линейной комбинации. Можно сказать, что вектор (1, 3, −7) имеет координаты (3, 2, −5) относительно базиса (1, 0, 1), (−1, −1, 0), (0, −1, 2).

То же верно для более высоких размерностей: если это возможно, мы можем записать вектор как линейную комбинацию других векторов, решив соответствующую систему линейных уравнений. Но не всегда возможно записать линейную комбинацию, и не всякая система линейных уравнений имеет единственное решение, а некоторые вообще не имеют решения. Вопрос о том, образует ли набор векторов базис, в вычислительном отношении эквивалентен вопросу о том, имеет ли система линейных уравнений единственное решение.

Эта глубокая связь — отличная заключительная мысль для части I, посвященной линейной алгебре. В книге будет представлено еще множество замечательных тем из линейной алгебры, но их ценность будет еще выше, если объединить их с основной темой части II — математическим анализом.

7.4.2. Упражнения

Упражнение 7.27. Как можно представить вектор (5, 5) в виде линейной комбинации векторов (10, 1) и (3, 2)?

Решение. Эта задача эквивалентна вопросу, какие числа a и b удовлетворяют уравнению

$$a\begin{pmatrix}10\\1\end{pmatrix}+b\begin{pmatrix}3\\2\end{pmatrix}=\begin{pmatrix}5\\5\end{pmatrix}$$

или какой вектор (a, b) удовлетворяет матричному уравнению

$$\begin{pmatrix}10 & 3\\1 & 2\end{pmatrix}\begin{pmatrix}a\\b\end{pmatrix}=\begin{pmatrix}5\\5\end{pmatrix}.$$

Мы можем найти решение с помощью NumPy:

```
>>> matrix = np.array(((10,3),(1,2)))
>>> vector = np.array((5,5))
>>> np.linalg.solve(matrix,vector)
array([-0.29411765, 2.64705882])
```

То есть искомая линейная комбинация имеет вид

$$-0{,}29411765\cdot\begin{pmatrix}10\\1\end{pmatrix}+2{,}64705882\cdot\begin{pmatrix}3\\2\end{pmatrix}=\begin{pmatrix}5\\5\end{pmatrix}.$$

Упражнение 7.28. Запишите вектор (3, 0, 6, 9) в виде линейной комбинации векторов (0, 0, 1, 1), (0, –2, –1, –1), (1, –2, 0, 2) и (0, 0, –2, 1).

Решение. Для этого необходимо решить систему линейных уравнений

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & -2 & -2 & 0 \\ 1 & -1 & 0 & -2 \\ 1 & -1 & 2 & 1 \end{pmatrix}\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} 3 \\ 0 \\ 6 \\ 9 \end{pmatrix},$$

где столбцы матрицы 4×4 — это векторы, на основе которых нужно построить линейную комбинацию. Библиотека NumPy дает нам решение этой системы:

```
>>> matrix = np.array(((0, 0, 1, 0), (0, -2, -2, 0), (1, -1, 0, -2),
(1, -1, 2, 1)))
>>> vector = np.array((3,0,6,9))
>>> np.linalg.solve(matrix,vector)
array([ 1., -3., 3., -1.])
```

То есть искомая линейная комбинация имеет вид:

$$1 \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} - 3 \cdot \begin{pmatrix} 0 \\ -2 \\ -1 \\ -1 \end{pmatrix} + 3 \cdot \begin{pmatrix} 1 \\ -2 \\ 0 \\ 2 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ -2 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 0 \\ 6 \\ 9 \end{pmatrix}.$$

КРАТКИЕ ИТОГИ ГЛАВЫ

- Объекты в двухмерной видеоигре можно выполнить в виде многоугольников, построенных из отрезков.
- Для двух векторов, **u** и **v**, все точки, определяемые формулой **u** + t**v**, где t — произвольное действительное число, лежат на прямой. Этой формулой можно описать любую прямую.
- Для любых действительных чисел a, b и c, где хотя бы одно число из пары a и b отлично от нуля, все точки (x, y) на плоскости, удовлетворяющие условию $ax + by = c$, лежат на прямой. Это называется *стандартной формой* уравнения прямой, и любую прямую можно записать в этой форме для некоторого набора чисел a, b и c. Уравнения прямых линий называются *линейными уравнениями*.
- Определение точки пересечения двух прямых на плоскости эквивалентно определению значений (x, y), удовлетворяющих одновременно двум

линейным уравнениям. Набор линейных уравнений, которые должны решаться одновременно, называется *системой линейных уравнений*.

- Решение системы двух линейных уравнений эквивалентно определению вектора, на который можно умножить известную матрицу 2×2, чтобы получить известный вектор.
- Библиотека NumPy имеет встроенную функцию `numpy.linalg.solve`, которая принимает матрицу и вектор и автоматически решает соответствующую систему линейных уравнений, если это возможно.
- Некоторые системы линейных уравнений не имеют решения. Например, если две прямые параллельны, они могут не иметь точек пересечения или таких точек может быть бесконечно много (такое возможно, когда прямые совпадают). Это означает, что не существует значения (x, y), которое одновременно удовлетворяет уравнениям обеих прямых. Матрица, представляющая такую систему, называется *сингулярной*.
- Плоскости в трехмерном пространстве являются аналогами линий в двухмерном пространстве. Это наборы точек (x, y, z), удовлетворяющих уравнениям вида $ax + by + cz = d$.
- Две непараллельные плоскости в трехмерном пространстве пересекаются в бесконечном количестве точек, в частности, набор точек пересечения образует одномерную прямую. Три плоскости могут иметь единственную точку пересечения, которую можно найти, решив систему трех линейных уравнений, представляющих плоскости.
- Прямые в двухмерном и плоскости в трехмерном пространстве являются частными случаями *гиперплоскостей* — наборов точек в n измерениях, которые являются решениями одного линейного уравнения.
- Чтобы найти единственное решение в n-мерном пространстве, нужна система не менее чем из n линейных уравнений. Если есть ровно n линейных уравнений и они имеют единственное решение, то такие уравнения называются *независимыми уравнениями*.
- Чтобы выяснить, как записать вектор в виде линейной комбинации заданного набора векторов, нужно решить систему линейных уравнений. Если набор векторов является базисом пространства, то решение есть всегда.

Часть II

Математический анализ и моделирование физического мира

В этой части мы приступаем к обзору математического анализа[1]. Математический анализ в широком смысле — это изучение непрерывных изменений, поэтому мы много будем говорить о том, как измерить скорость изменения различных величин и что эти скорости изменения могут нам сказать.

На мой взгляд, математический анализ заслужил репутацию сложного предмета из-за необходимости выполнять множество алгебраических вычислений, а не из-за незнакомых понятий. Если вы когда-нибудь владели автомобилем или водили его, то имеете интуитивное представление о скоростях и суммарных значениях: спидометр измеряет *скорость* движения с течением времени, а одометр — суммарное количество пройденных миль. В какой-то степени их показания должны совпадать. Если спидометр показывает более высокое значение в течение некоторого времени, то цифра на одометре в течение этого же времени должна увеличиться на большую величину, и наоборот.

В матанализе, как вы узнаете в дальнейшем, если есть функция, дающая суммарное значение в любой момент времени, то можно вычислить скорость его изменения как функцию времени. Операция взятия суммарной функции и получения на ее основе функции скорости называется *производной*. Точно так же, имея функцию скорости, можно восстановить соответствующую суммарную функцию. Эта операция называется *интегралом*. Всю главу 8 мы посвятим осмыслению этих преобразований путем применения их к измеренному объему жидкости (суммарная функция) и расходу (соответствующая функция скорости). В главе 9 распространим эти понятия на несколько измерений. Для моделирования

---

[1] В англо-американской традиции классическому математическому анализу соответствуют программы курсов с наименованием «исчисление» (англ. Calculus). — *Примеч. пер.*

движущегося объекта в видеоигре нам нужно рассмотреть взаимосвязь между скоростью и положением по каждой координате независимо.

Получив базовое представление о матанализе в главах 8 и 9, мы перейдем к знакомству с механикой в главе 10. Она доставит вам больше удовольствия, чем теория математического анализа, потому что большую часть работы мы сможем переложить на Python. Здесь мы смоделируем математические выражения как маленькие компьютерные программы, благодаря чему получим возможность анализировать и преобразовывать их для определения производных и интегралов. То есть в главе 10 будет показан совершенно другой подход к выполнению математических операций в программах, который называется *символьным программированием*.

В главе 11 мы вернемся к многомерному матанализу. Скорость на спидометре или расход жидкости, протекающей через трубу, — это функции, меняющиеся во времени, но у нас могут быть и функции, меняющиеся в пространстве. Они принимают векторы на входе и возвращают числа или векторы на выходе. Например, представление силы гравитации как функции в двухмерном пространстве позволит добавить сходство с физическим миром в видеоигру из главы 7. Ключевой операцией матанализа для функций, меняющихся в пространстве, является *градиент* — операция, сообщающая пространственное направление, в котором функция возрастает с наибольшей скоростью. Поскольку градиент измеряет скорость, он подобен векторной версии обычной производной. В главе 12 мы используем градиент для *оптимизации* функции, то есть для поиска входных значений, для которых она возвращает наибольший результат. Следуя направлению вектора градиента, можно находить все более высокие выходные значения и в итоге достичь максимального значения для всей функции.

В главе 13 мы рассмотрим совершенно другое применение математического анализа. Оказывается, интеграл функции может многое сказать о геометрии графика функции. В частности, интегрируя произведение двух функций, можно получить оценку сходства их графиков. Мы применим этот вид анализа к звуковым волнам. *Звуковая волна* — это график функции, описывающей звук, и по его форме можно определить, звук громкий или тихий, высокий или низкий и т. д. Сравнивая звуковую волну с различными музыкальными нотами, мы сможем узнать, какие из них она содержит. Представление о звуковой волне как о функции соответствует важному математическому понятию — *рядам Фурье*.

По сравнению с частью I эта часть больше похожа на шведский стол, потому что рассматривает две основные темы, на которые стоит обратить внимание. Во-первых, это понятие скорости изменения функции: возрастание или убывание функции в данной точке говорит нам, как найти большее или меньшее значение. Во-вторых, это понятие операции, которая принимает функции на входе и возвращает функции на выходе. В матанализе ответ на многие вопросы дается в виде функции. Эти две темы будут ключевыми для приложений машинного обучения, которыми мы займемся в части III.

8

Скорость изменения

В этой главе

- ✓ Вычисление средней скорости изменения математической функции.
- ✓ Аппроксимация мгновенной скорости изменения в точке.
- ✓ Представление об изменении скорости изменений.
- ✓ Восстановление функции по скорости ее изменения.

В этой главе я познакомлю вас с двумя наиболее важными понятиями математического анализа — производной и интегралом. Обе эти операции связаны с функциями. *Производная* получает функцию и возвращает другую функцию, измеряющую скорость ее изменения. *Интеграл*, наоборот, получает функцию, представляющую скорость изменения, и возвращает функцию, измеряющую исходное суммарное значение.

Рассмотрим простой пример из моей практики анализа данных о добыче нефти. Представьте насос, поднимающий сырую нефть из скважины и подающий ее по трубе в резервуар. Труба оборудована датчиком-расходомером, непрерывно измеряющим скорость потока жидкости, а резервуар — датчиком-уровнемером, определяющим уровень жидкости в резервуаре и сообщающим объем нефти, находящейся внутри (рис. 8.1).

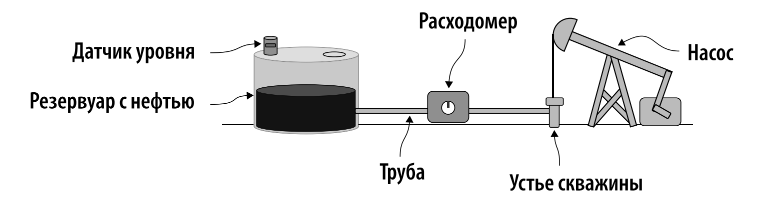


Рис. 8.1. Схематическое изображение насоса, поднимающего нефть из скважины в резервуар

Показания датчика уровня сообщают объем нефти в резервуаре в виде функции от времени, а показания расходомера сообщают объем нефти, поступающей в резервуар в единицах объема в час и тоже в виде функции от времени. В этом примере объем является суммарным значением, а расход — скоростью его изменения.

В этой главе мы будем решать две основные задачи. Во-первых, начав с известных суммарных объемов в разные моменты времени, вычислим расход как функцию от времени, используя производную. Во-вторых, решим противоположную задачу: начав с расхода как функции от времени, вычислим суммарный объем нефти в резервуаре в заданный момент времени с помощью интеграла. Этот процесс показан на рис. 8.2.

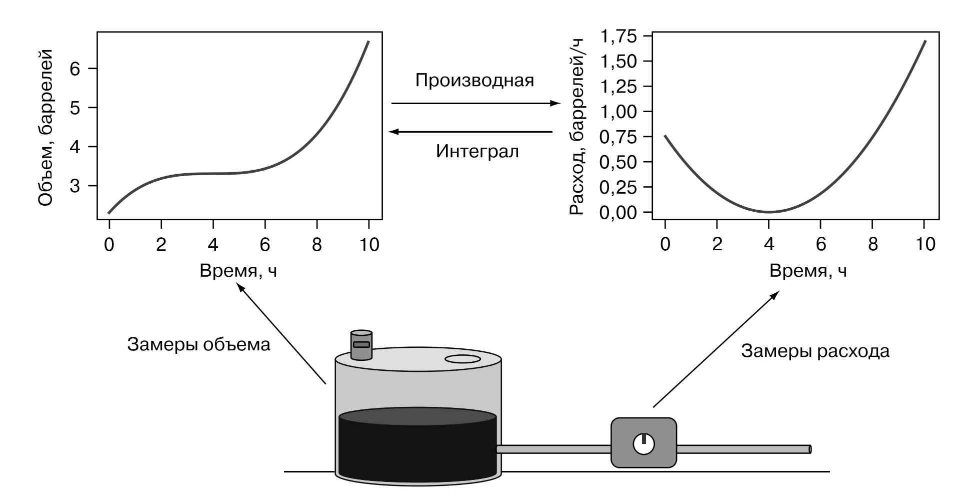


Рис. 8.2. Определение расхода, исходя из объема, с применением производной, а затем определение объема на основе расхода с помощью интеграла

Мы напишем функцию `get_flow_rate(volume_function)`, которая принимает функцию объема и возвращает новую функцию, вычисляющую расход в данный момент времени. Затем напишем вторую функцию, `get_volume(flow_rate_function)`,

которая принимает функцию расхода и возвращает функцию, дающую объем в данный момент времени. Попутно я добавлю несколько небольших примеров для разминки, чтобы направить ваши размышления о скорости изменений в правильное русло.

Несмотря на то что основные идеи матанализа не особенно сложны, он считается трудным предметом, потому что требует большого объема утомительных алгебраических вычислений. Поэтому в данной главе я сосредоточусь на представлении новых тем и покажу не так уж много новых методов. Для решения большинства примеров достаточно будет знаний линейной алгебры, которые вы приобрели в главе 7. Приступим!

8.1. ВЫЧИСЛЕНИЕ СРЕДНЕГО РАСХОДА ПО ОБЪЕМУ

Для начала предположим, что нам известен объем нефти в резервуаре в каждый момент времени, а узнать его можно с помощью функции `volume` на Python. Эта функция принимает время в часах, прошедшее с некоторого начального момента, и возвращает объем нефти в резервуаре в этот момент, измеряемый в единицах, называемых баррелями. Чтобы сосредоточиться на сути, а не на алгебре, я даже не буду раскрывать формулу функции (те, кому это интересно, могут увидеть ее в примерах к книге). Прямо сейчас вам нужно лишь несколько раз вызвать ее и построить график. Проделав это, вы должны получить график, изображенный на рис. 8.3.

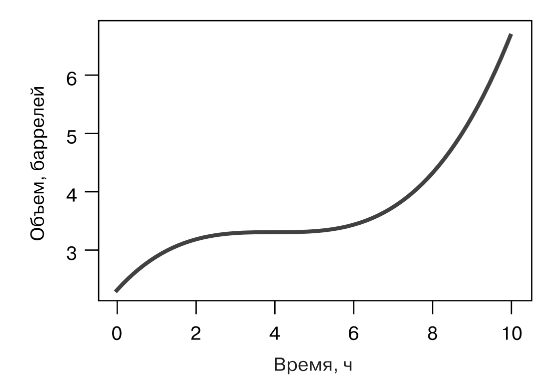


Рис. 8.3. График функции volume, показывающий изменение объема нефти в резервуаре с течением времени

Наша текущая задача — определение скорости поступления нефти в резервуар в любой момент времени, поэтому начнем с того, что рассчитаем ее интуитивным способом. Для этого напишем функцию `average_flow_rate(v, t1, t2)`, которая принимает функцию объема `v`, время начала `t1` и время окончания `t2`

и возвращает число, представляющее *среднюю скорость поступления* нефти в резервуар за заданный интервал времени. То есть она должна сообщить нам среднюю скорость в баррелях в час, с какой нефть поступала в резервуар в течение этого времени.

8.1.1. Реализация функции average_flow_rate

Предлог «в» в единице измерения «баррели в час» предполагает выполнение деления для получения ответа. Чтобы найти средний расход (скорость поступления), нужно разделить значение изменения объема на прошедшее время:

$$\textbf{средний расход} = \frac{\textbf{изменение объема}}{\textbf{прошедшее время}}.$$

Прошедшее время — это интервал между временем начала t_1 и окончания t_2, измеренный в часах, который вычисляется как $t_2 - t_1$. Если есть функция $V(t)$, которая сообщает объем как функцию от времени, то общее изменение объема равно разности объемов в моменты t_2 и t_1, или $V(t_2) - V(t_1)$. Это дает нам более конкретное уравнение

$$\textbf{средний расход в интервале между } t_1 \textbf{ и } t_2 = \frac{V(t_2) - V(t_1)}{t_2 - t_1}.$$

Именно так рассчитывается скорость изменений в различных контекстах. Например, скорость движения автомобиля — это расстояние, преодолеваемое за единицу времени. Чтобы вычислить среднюю скорость за поездку, нужно разделить пройденное в целом расстояние в милях на затраченное время в часах, получив скорость в милях в час. Чтобы узнать пройденное расстояние и затраченное время, нужно свериться с часами и одометром в начале и конце поездки.

Формула вычисления среднего расхода зависит от функции объема V и времени начала t_1 и конца t_2, которые можно передать соответствующей функции на Python с нужными параметрами. Тело функции — прямой перевод этой математической формулы на Python:

```
def average_flow_rate(v,t1,t2):
    return (v(t2) - v(t1))/(t2 - t1)
```

Это простая функция, но она важна, поэтому нужно уделить ей внимание. Воспользуемся функцией `volume` (график которой изображен на рис. 8.3, а исходный код включен в набор примеров) и попробуем узнать среднюю скорость поступления нефти в резервуар между 4- и 9-часовой отметками. В этом случае $t_1 = 4$, $t_2 = 9$. Чтобы найти начальный и конечный объемы, вызовем функцию `volume` с этими параметрами:

```
>>> volume(4)
3.3
>>> volume(9)
5.253125
```

Округлив для простоты полученные значения, находим разницу между двумя объемами, 5,25 – 3,3 = 1,95 барреля, а общее истекшее время составляет 9 – 4 = 5 часов. Таким образом, средний расход составляет примерно 1,95 барреля, деленные на 5 часов, или 0,39 барреля в час. Функция подтверждает, что мы все сделали правильно:

```
>>> average_flow_rate(volume,4,9)
0.390625
```

На этом завершаем первый простой пример определения скорости изменения функции. Это было не так уж сложно! Но прежде чем перейти к более интересным примерам, потратим еще немного времени на интерпретацию того, что делает функция вычисления объема.

8.1.2. Изображение среднего расхода секущей прямой

Еще один полезный способ оценить среднюю скорость изменения объема с течением времени — посмотреть на график объема. Сосредоточимся на двух точках на графике, между которыми мы рассчитали средний расход. На рис. 8.4 я соединил их прямой. Прямая, проходящая через две точки на таком графике, называется *секущей*.

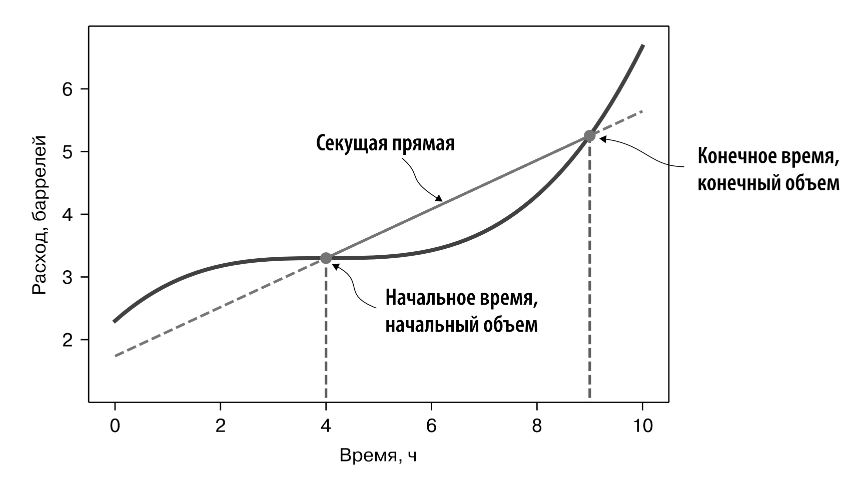


Рис. 8.4. Секущая прямая соединяет начальную и конечную точки на графике изменения объема

Как видите, отметка на графике, соответствующая 9 часам, находится выше отметки, соответствующей 4 часам, потому что за этот период объем нефти в резервуаре увеличился. В результате секущая прямая, соединяющая начальную и конечную точки, направлена вверх. Наклон секущей *напрямую* связан со средней величиной расхода на временном интервале, и вот почему. Наклон прямой, соединяющей две точки, пропорционален величине изменения вертикальной координаты, деленной на величину изменения горизонтальной координаты. При этом вертикальная координата изменяется от $V(t_1)$ до $V(t_2)$ на величину $V(t_2) - V(t_1)$, а горизонтальная координата изменяется от t_1 до t_2 на величину $t_2 - t_1$. Затем $V(t_2) - V(t_1)$ делится на $t_2 - t_1$, в точности как вычисляется средний расход (рис. 8.5)!

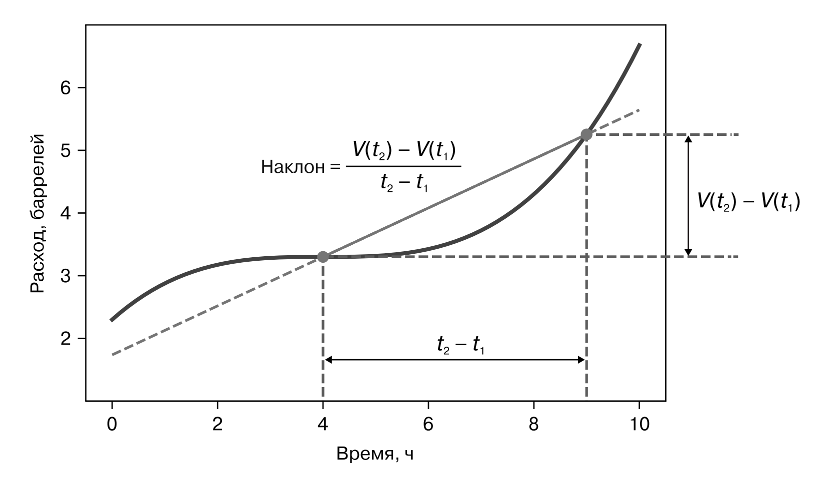


Рис. 8.5. Наклон секущей прямой вычисляется так же, как средняя скорость изменения функции volume

Рассуждая о средней скорости изменения функции, вы можете сами рисовать секущие прямые на графике.

8.1.3. Отрицательные скорости изменения

Следует упомянуть еще один случай, когда секущая прямая имеет отрицательный наклон. На рис. 8.6 показан график другой функции объема, которую можно найти в примерах к этой книге под именем `decreasing_volume`. На рис. 8.6 показан график уменьшения объема нефти в резервуаре с течением времени.

Этот пример противоречит предыдущему примеру, потому что мы не ожидаем, что нефть будет вытекать из резервуара обратно в землю. И все же он показывает, что секущая может быть направлена вниз, например, на интервале от $t = 0$ до $t = 4$. В этом временном промежутке изменение объема нефти составило –3,2 барреля (рис. 8.7).

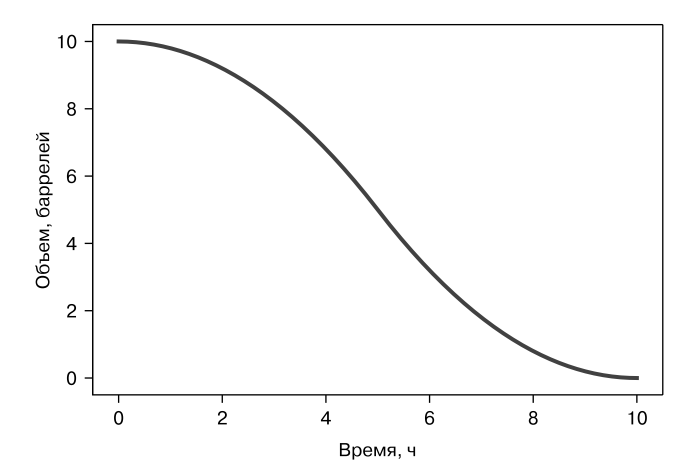


Рис. 8.6. Другая функция объема показывает, что объем нефти в резервуаре со временем уменьшается

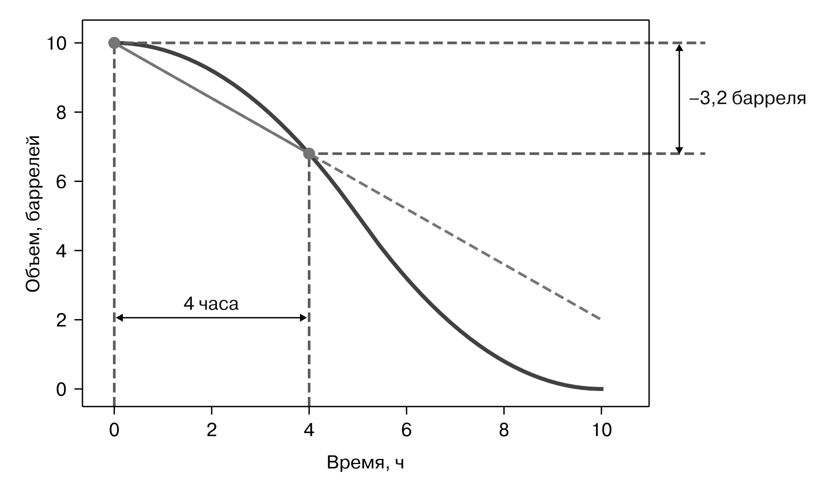


Рис. 8.7. Две точки на графике, определяющие секущую прямую с отрицательным наклоном

В этом случае наклон составляет –3,2 барреля, деленные на 4 часа, или –0,8 барреля в час. Это означает, что скорость поступления нефти в резервуар составляет –0,8 барреля в час. Правильнее было бы сказать, что нефть *вытекает* из резервуара со скоростью 0,8 барреля в час. Независимо от того, увеличивается или уменьшается функция объема, функция `average_flow_rate` надежно вычисляет скорость ее изменения. В данном случае

```
>>> average_flow_rate(decreasing_volume,0,4)
-0.8
```

Получив функцию вычисления скорости изменения объема, мы готовы сделать еще один шаг в следующем разделе и выяснить, как скорость изменяется во времени.

8.1.4. Упражнения

Упражнение 8.1. Представьте, что вы отправились в поездку в полдень, когда одометр показывал общий пробег 77 641 милю, и закончили ее в 16:30, когда одометр показывал 77 905 миль. Вычислите среднюю скорость за все время поездки.

Решение. Поездка продлилась 4,5 часа, пройденное расстояние составило 77 905 – 77 641 = 264 мили. Отсюда средняя скорость равна 264 мили/4,5 часа, или примерно 58,7 миль/ч.

Упражнение 8.2. Напишите функцию `secant_line(f, x1, x2)`, которая принимает функцию `f(x)` и два значения, `x1` и `x2`, и возвращает новую функцию, представляющую секущую прямую во времени. Например, после выполнения инструкции `line = secant_line(f,x1,x2)` вызов `line(3)` должен вернуть значение y секущей прямой, соответствующее $x = 3$.

Решение

```
def secant_line(f,x1,x2):
    def line(x):
        return f(x1) + (x-x1) * (f(x2)-f(x1))/(x2-x1)
    return line
```

Упражнение 8.3. Напишите функцию, которая использует код из предыдущего упражнения и рисует секущую прямую для функции `f`, соединяющую две заданные точки.

Решение

```
def plot_secant(f,x1,x2,color='k'):
    line = secant_line(f,x1,x2)
    plot_function(line,x1,x2,c=color)
    plt.scatter([x1,x2],[f(x1),f(x2)],c=color)
```

8.2. ГРАФИК ЗАВИСИМОСТИ СРЕДНЕЙ СКОРОСТИ ОТ ВРЕМЕНИ

Одна из наших главных целей в этой главе — на основе функции объема восстановить функцию скорости поступления нефти. Чтобы выразить скорость поступления в виде функции от времени, нужно узнать, как быстро меняется объем нефти в резервуаре в разные моменты. Во-первых, как показано на рис. 8.8, скорость поступления меняется во времени — разные секущие на графике объема имеют разный наклон.

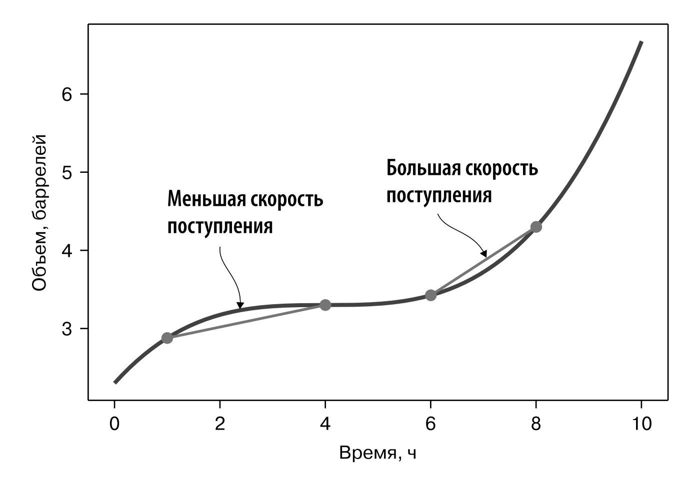


Рис. 8.8. Разные секущие прямые на графике объема имеют разный наклон, что говорит об изменении скорости поступления нефти

В этом разделе мы приблизимся к определению расхода как функции от времени путем вычисления среднего расхода на разных интервалах. Для этого разобьем 10-часовой период на несколько меньших интервалов фиксированной продолжительности (например, 10 интервалов по 1 часу) и вычислим средний расход для каждого.

Мы упакуем эти вычисления в функцию `interval_flow_rates(v, t1, t2, dt)`, где `v` — функция объема, `t1` и `t2` — время начала и конца, а `dt` — фиксированная продолжительность временных интервалов. Эта функция будет возвращать список пар времени и расхода. Например, если мы разобьем 10 часов на сегменты по 1 часу, то результат должен выглядеть так:

```
[(0,...), (1,...), (2,...), (3,...), (4,...), (5,...), (6,...), (7,...),
 (8,...), (9,...)]
```

На месте многоточий `...` будут находиться значения расхода в соответствующие часы. Получив эти пары, мы сможем изобразить их в виде точечной диаграммы рядом с функцией расхода из начала главы и сравнить результаты.

8.2.1. Определение среднего расхода в разные промежутки времени

В качестве первого шага реализации `interval_flow_rates()` найдем начальные точки для каждого временного интервала. Это означает, что мы должны создать список значений времени от начального момента `t1` до момента окончания `t2` с шагом, равным длине интервала `dt`.

В библиотеке NumPy есть удобная функция `arange`, которая делает эту работу. Например, если передать ей интервал времени от 0 до 10 часов и задать продолжительность одного шага равной 0,5 часа, то она вернет следующий список значений, соответствующих началу каждого шага:

```
>>> import numpy as np
>>> np.arange(0,10,0.5)
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. , 5.5, 6. ,
       6.5, 7. , 7.5, 8. , 8.5, 9. , 9.5])
```

Обратите внимание на то, что отметка 10 часов, соответствующая времени окончания, не включена в список. Это связано с тем, что в список включаются отметки времени, соответствующие началу каждого получасового шага, а интервал от $t = 10$ до $t = 10{,}5$ не является частью общего временного интервала, который мы задали.

Прибавив продолжительность интервала `dt` к времени начала, можно получить время конца этого интервала. Например, интервал, начинающийся в 3,5 часа, заканчивается в 3,5 + 0,5 = 4 часа. Чтобы реализовать функцию `interval_flow_rates`, нужно просто вызвать функцию `average_flow_rate` для каждого интервала. Вот как выглядит законченная функция:

```
def interval_flow_rates(v,t1,t2,dt):
    return [(t,average_flow_rate(v,t,t+dt))
            for t in np.arange(t1,t2,dt)]
```

Для каждого интервала, начинающегося в момент времени t, вычисляется средний расход в период между t и t + dt. (Для этого нужен список пар t с соответствующим расходом.)

Если передать в вызов `interval_flow_rates` функцию `volume`, 0 и 10 часов в качестве времени начала и конца общего интервала и величину шага, равную 1 часу, то в ответ мы получим список, сообщающий скорость поступления нефти в каждый час:

```
>>> interval_flow_rates(volume,0,10,1)
[(0, 0.578125),
 (1, 0.296875),
 (2, 0.109375),
 (3, 0.015625),
 (4, 0.015625),
 (5, 0.109375),
```

```
(6, 0.296875),
(7, 0.578125),
(8, 0.953125),
(9, 1.421875)]
```

Глядя на этот список, можно подметить несколько аспектов. Средняя скорость поступления нефти всегда положительная, а это означает, что каждый час ее объем в резервуаре только увеличивался. Скорость поступления нефти снижается до минимального значения примерно через 3 и 4 часа, а затем увеличивается до максимального значения в последний час. Это станет еще очевиднее, если нанести скорость на график.

8.2.2. График интервальных расходов

Быстро построить график изменения расхода с течением времени можно с помощью функции `scatter` из библиотеки Matplotlib. Эта функция принимает два отдельных списка с горизонтальными и вертикальными координатами и рисует набор точек на графике. Чтобы воспользоваться ею, мы должны создать два списка с 10 значениями времени и расхода, а затем передать их функции. Чтобы не повторять процесс, объединим все в одну функцию:

```
def plot_interval_flow_rates(volume,t1,t2,dt):
    series = interval_flow_rates(volume,t1,t2,dt)
    times = [t for (t,_) in series]
    rates = [q for (_,q) in series]
    plt.scatter(times,rates)
```

Вызов `plot_interval_flow_rates(volume,0,10,1)` создаст точечную диаграмму на основе данных, полученных с помощью `interval_flow_rates`. На рис. 8.9 показан результат построения графика функции `volume` от нуля до 10 часов с шагом в 1 час.

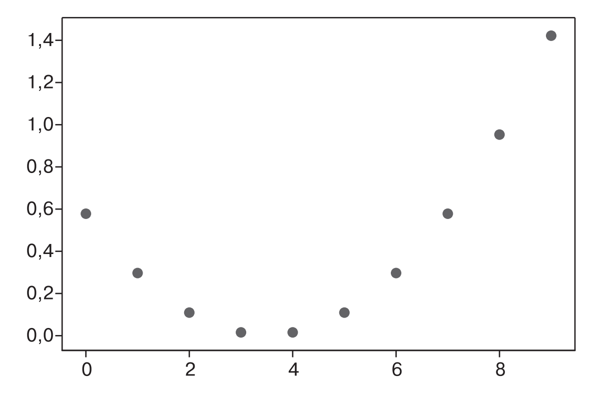


Рис. 8.9. График изменения среднего расхода с течением времени

Этот график подтверждает наши умозаключения, сделанные на основе данных: средний расход уменьшается до самого малого значения на отметках 3 и 4 часа, а затем снова увеличивается до самого высокого значения, достигая почти 1,5 барреля в час. Сравним средние значения с фактической функцией расхода. Я не хочу грузить вас формулой зависимости расхода от времени, поэтому не буду приводить ее здесь. Но я включил функцию `flow_rate` в примеры с исходным кодом для этой книги, и мы можем построить ее график рядом с точечной диаграммой (рис. 8.10).

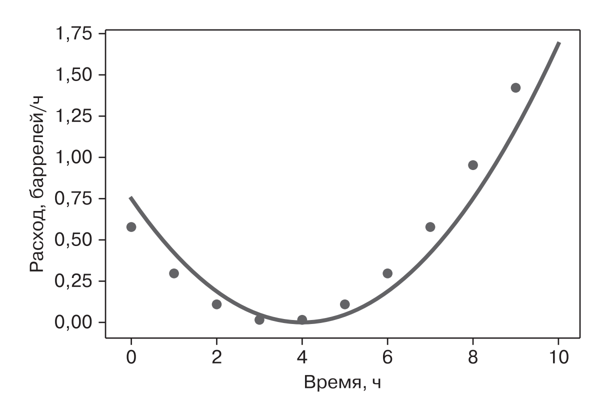


Рис. 8.10. График изменения среднего расхода *(точки)* и фактического расхода *(сплошная кривая)*

Эти два графика отражают один и тот же процесс, но не совсем точно совпадают. Причина в том, что точечная диаграмма отражает средний расход, тогда как функция `flow_rate` показывает *мгновенное* значение расхода в любой момент времени.

Чтобы понять разницу, вспомним пример с поездкой на автомобиле. Если вы преодолели 60 миль за 1 час, то средняя скорость составит 60 миль/ч. Однако маловероятно, что в течение этого часа спидометр постоянно показывал ровно 60 миль/ч. В какой-то момент на прямых участках ваша *мгновенная скорость* могла достигать 70 миль/ч, а на извилистых или загруженных участках вы могли снизить ее до 50 миль/ч.

Точно так же показания расходомера на трубопроводе не обязательно должны согласовываться со средним расходом за текущий час. Получается, что если сделать временные интервалы меньше, то графики должны быть ближе друг к другу. На рис. 8.11 показаны графики функции мгновенного расхода и среднего расхода с 20-минутными интервалами (1/3 часа).

Средние значения расхода все еще неточно соответствуют мгновенному расходу, но теперь два графика намного ближе друг к другу. В следующем разделе мы воспользуемся этим приемом: вычислим средние значения расхода на очень

маленьких интервалах и увидим, что при таких условиях разница между средним и мгновенным расходом практически незаметна.

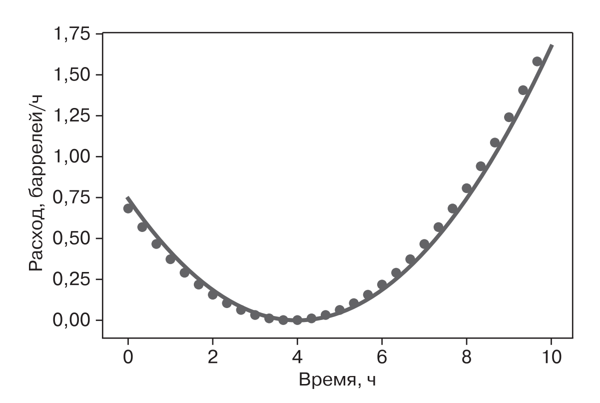
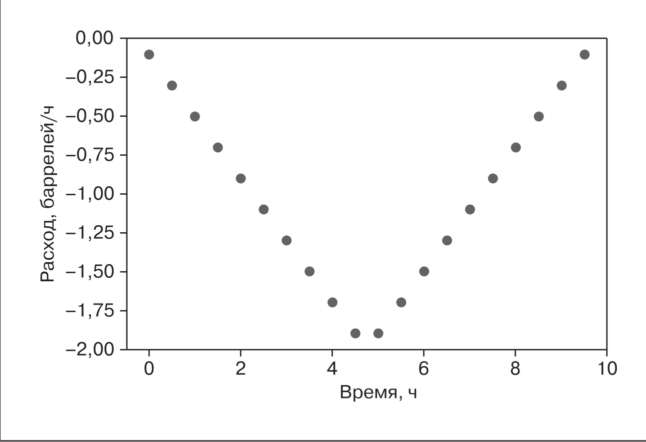


Рис. 8.11. Графики изменения расхода с течением времени и изменения среднего расхода с 20-минутными интервалами

8.2.3. Упражнения

Упражнение 8.4. Постройте график изменения расхода с течением времени, используя `decreasing_volume` и получасовые интервалы. Когда расход самый низкий? То есть когда нефть вытекает из бака с наибольшей скоростью?

Решение. Запустив `plot_interval_flow_rates(decreasing_volume,0, 10,0.5)`, можно увидеть, что самый низкий расход (наибольшее по абсолютной величине отрицательное значение) — около отметки 5 часов.

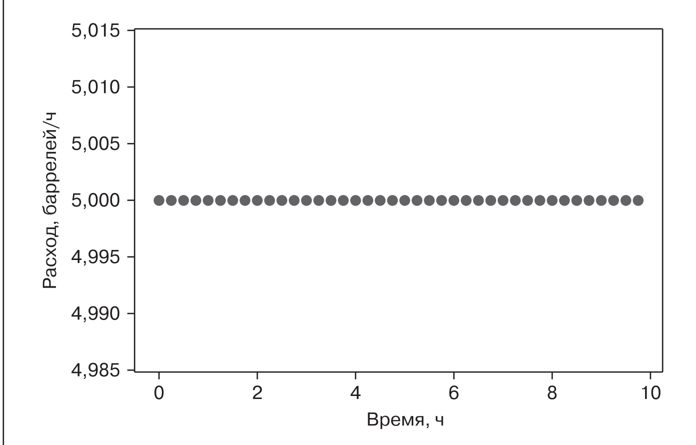


Упражнение 8.5. Напишите функцию `linear_volume_function` и постройте график зависимости расхода от времени, чтобы показать, что расход не меняется.

Решение. Функция `linear_volume_function(t)` выполняет вычисления по формуле $V(t) = at + b$, где a и b — константы, например:

```
def linear_volume_function(t):
    return 5*t + 3

plot_interval_flow_rates(linear_volume_function,0,10,0.25)
```



Этот график показывает, что для линейной функции изменения объема значение расхода постоянно во времени.

8.3. АППРОКСИМАЦИЯ ЗНАЧЕНИЙ МГНОВЕННОГО РАСХОДА

Вычисляя среднюю скорость изменения функции объема с использованием все меньших и меньших интервалов времени, мы все ближе и ближе подходим к мгновенным значениям. Но если попытаться вычислить среднюю скорость изменения объема на интервале, начальное время которого совпадает с конечным, мы столкнемся с проблемой. В момент времени t формула вычисления среднего расхода будет выглядеть так:

$$\textbf{Средний расход в момент времени } t = \frac{V(t)-V(t)}{t-t} = \frac{0}{0}.$$

Результат операции деления 0/0 не определен, поэтому такой способ оценки мгновенной скорости не работает. Здесь алгебра не поможет, и нам следует обратиться к рассуждениям из области исчисления. В матанализе есть операция, называемая *производной*, которая решает проблему неопределенности деления и сообщает мгновенную скорость изменения функции.

В этом разделе я объясню, почему функция мгновенного расхода, которая в матанализе называется *производной* функции объема, четко определена и как ее аппроксимировать. Мы напишем функцию `instantaneous_flow_rate(v,t)`, которая принимает функцию объема v и единственную отметку времени t и возвращает приблизительную мгновенную скорость, с которой нефть поступает в резервуар. Результат измеряется в баррелях в час и должен точно соответствовать значению функции `instantaneous_flow_rate`.

Затем напишем вторую функцию `get_flow_rate_function(v)`, которая является каррированной версией `instantaneous_flow_rate()`. Она будет принимать функцию объема и возвращать функцию, которая принимает время и возвращает мгновенный расход. Получение этой функции — первая из двух основных целей этой главы: начать с функции объема и создать соответствующую функцию расхода.

8.3.1. Определение наклона секущих прямых на коротких интервалах

Прежде чем приступить к программированию, я хочу объяснить, почему имеет смысл говорить о мгновенном расходе. Для этого увеличим график изменения объема и посмотрим, что мы имеем (рис. 8.12). Выберем точку $t = 1$ час и рассмотрим ее ближайшие окрестности.

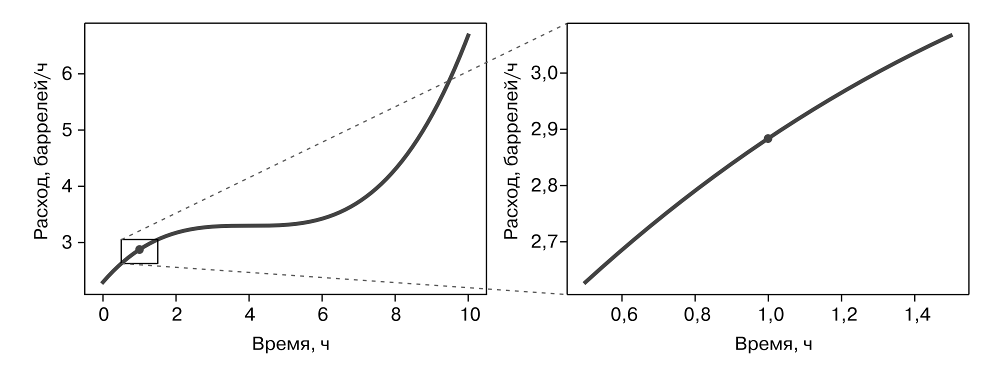


Рис. 8.12. Увеличение окна протяженностью в 1 час с точкой $t = 1$ в середине

На этом коротком временном интервале кривизна графика изменения объема почти не видна. То есть график в этом окне имеет меньшую изменчивость, чем на

всем 10-часовом интервале. Мы можем убедиться в этом, начертив несколько секущих прямых и увидев, что они имеют примерно одинаковый наклон (рис. 8.13).

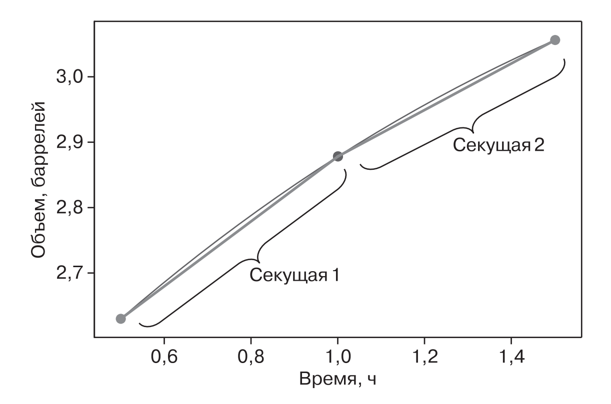


Рис. 8.13. Две секущие прямые, пересекающиеся на отметке 1 час, имеют почти одинаковый наклон, а это означает, что расход мало меняется на этом интервале времени

Если увеличить масштаб еще больше, график будет выглядеть еще менее изменчивым. В окне, включающем интервал от 0,9 до 1,1 часа, график объема выглядит почти как прямая линия. Если провести секущую прямую через этот интервал, то подъем графика над секущей будет почти незаметен (рис. 8.14).

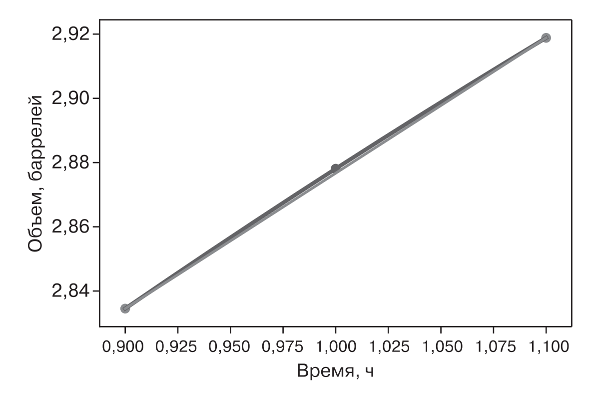


Рис. 8.14. На еще более коротком интервале график изменения объема около отметки 1 час выглядит почти как прямая

Наконец, если еще увеличить масштаб, чтобы окно охватывало интервал между $t = 0{,}99$ и $t = 1{,}01$ часа, то график изменения объема будет неотличим от прямой

линии (рис. 8.15). На этом уровне кажется, что секущая прямая точно соответствует графику функции, который выглядит как прямая.

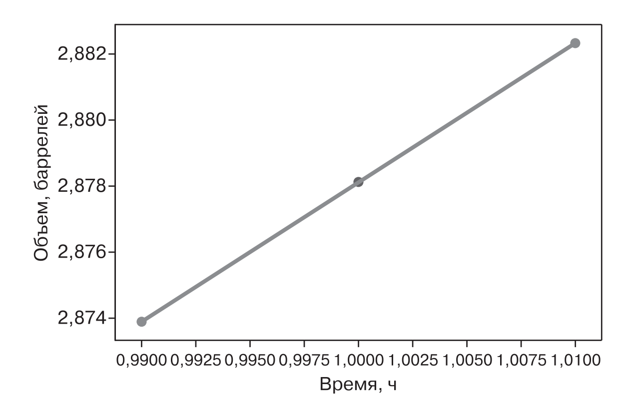


Рис. 8.15. Если увеличить масштаб еще больше, то график изменения объема становится визуально неотличимым от прямой

Если продолжить увеличивать масштаб, график будет все больше приближаться к прямой линии. Дело не в том, что в этой точке он представляет собой прямую, а в том, что становится все ближе и ближе к прямой. Из предшествующих рассуждений следует вывод, что в любой точке графика гладкой функции, такой как функция объема, существует единственная прямая, наилучшим образом аппроксимирующая его. Следующие вычисления показывают, что наклоны секущих прямых на все более коротких интервалах сходятся к одному значению, а это означает, что мы действительно приближаемся к единственному наилучшему приближению наклона:

```
>>> average_flow_rate(volume,0.5,1.5)
0.42578125
>>> average_flow_rate(volume,0.9,1.1)
0.4220312499999988
>>> average_flow_rate(volume,0.99,1.01)
0.42187656249998945
>>> average_flow_rate(volume,0.999,1.001)
0.42187501562509583
>>> average_flow_rate(volume,0.9999,1.0001)
0.42187500015393936
>>> average_flow_rate(volume,0.99999,1.00001)
0.4218750000002602
```

Отбросив практически ничего не значащие нули, мы получаем число, к которому приближаемся: 0,421875 барреля в час. Можно сделать вывод, что прямая наилучшего приближения для функции объема в точке $t = 1$ час имеет наклон

0,421875. Если снова уменьшить масштаб (рис. 8.16), то можно увидеть, как выглядит эта линия наилучшего приближения.

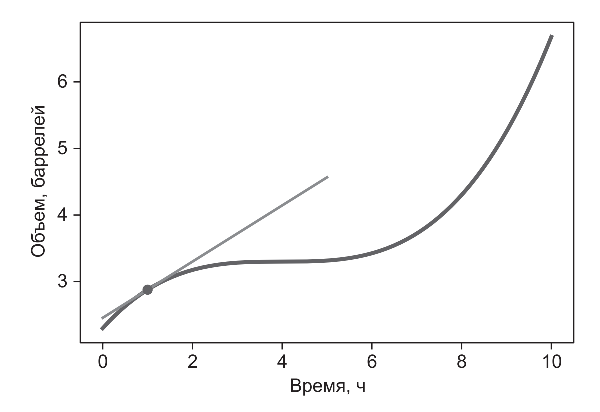


Рис. 8.16. Прямая с наклоном 0,421875 — это наилучшее приближение функции объема в точке $t = 1$ час

Эта прямая называется *касательной* к графику объема в точке $t = 1$ час и отличается тем, что в этой точке она соприкасается с кривой графика объема. Поскольку касательная — это прямая, которая лучше всего аппроксимирует график объема, ее наклон — наилучшая мера мгновенного наклона графика и, следовательно, мгновенного расхода в точке $t = 1$. А теперь самое интересное: функция `flow_rate`, которую я включил в примеры исходного кода, дает нам точно такое же число, к которому приближается наклон секущей с увеличением масштаба:

```
>>> flow_rate(1)
0.421875
```

Чтобы иметь касательную, функция должна быть гладкой. В качестве упражнения в конце этого раздела вам будет предложено проделать то же самое с негладкой функцией, и вы увидите, что она не имеет прямой наилучшего приближения. Если есть возможность найти касательную к графику функции в некоторой точке, то ее наклон называется *производной функции в этой точке*. Например, производная функции объема в точке $t = 1$ равна 0,421875 барреля в час.

8.3.2. Построение функции мгновенного расхода

Теперь, когда мы узнали, как рассчитать мгновенную скорость изменения функции объема, у нас есть все, что нужно для реализации функции `instantaneous_flow_rate`. Однако предстоит преодолеть серьезное препятствие, стоящее на пути к автоматизации процедуры, которую мы использовали, а именно: Python не может на глазок определить наклон нескольких небольших секущих прямых и решить,

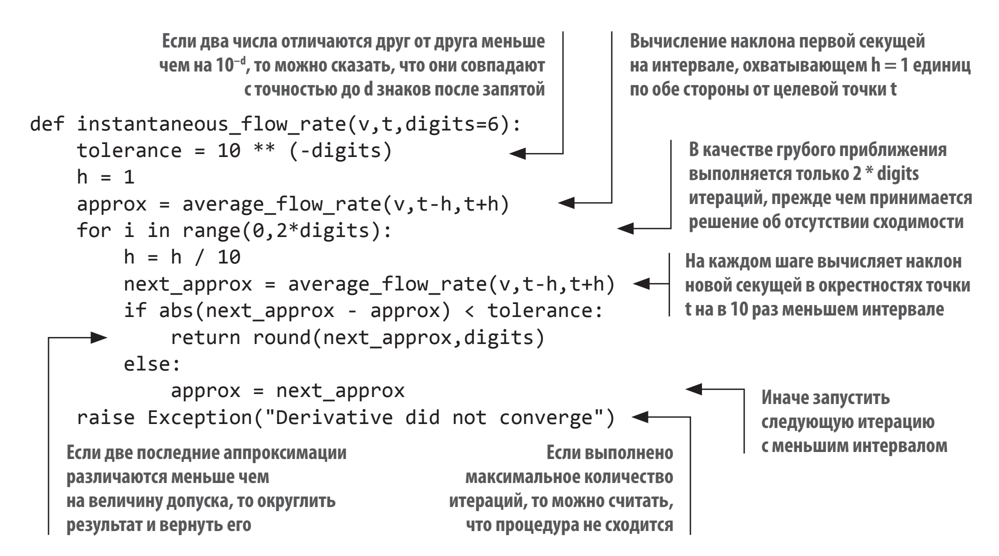
к какому числу они сходятся. Чтобы обойти это препятствие, можно пойти другим путем и вычислять наклон секущих для все меньших и меньших интервалов, пока он не стабилизируется на величине с некоторой заданной точностью.

Например, мы могли бы находить наклон секущих прямых на интервалах, последовательно уменьшающихся в 10 раз, пока результат не стабилизируется на уровне первых четырех знаков после запятой. В следующей таблице снова показаны значения наклона.

Интервал	Наклон секущей
0,5–1,5	0,42578125
0,9–1,1	0,4220312499999988
0,99–1,01	0,42187656249998945
0,999–1,001	0,42187501562509583

В последних двух строках наклон совпадает с точностью до четырех знаков после запятой (они различаются менее чем на 10^{-4}), поэтому можно округлить полученное значение до 0,4219 и назвать его окончательным результатом. Это не точный результат 0,421875, но достаточно близкая аппроксимация по выбранному количеству знаков после запятой.

Зафиксировав количество цифр в аппроксимации, мы получили способ, позволяющий определить достижение результата. Если после некоторого большого количества шагов результат так и не сошелся к заданному количеству цифр, то можно сказать, что прямой наилучшего приближения нет, а значит, нет и производной в точке. Вот как эта процедура переводится на язык Python:




```
def instantaneous_flow_rate(v,t,digits=6):
    tolerance = 10 ** (-digits)
    h = 1
    approx = average_flow_rate(v,t-h,t+h)
    for i in range(0,2*digits):
        h = h / 10
        next_approx = average_flow_rate(v,t-h,t+h)
        if abs(next_approx - approx) < tolerance:
            return round(next_approx,digits)
        else:
            approx = next_approx
    raise Exception("Derivative did not converge")
```

Я произвольно выбрал точность по умолчанию в шесть знаков, поэтому данная функция соответствует нашему результату для мгновенного расхода на отметке 1 час:

```
>>> instantaneous_flow_rate(volume,1)
0.421875
```

Теперь мы можем вычислить мгновенный расход в любой момент времени, а это значит, что у нас есть полные данные функции расхода. Затем можно построить график и убедиться, что он соответствует функции `flow_rate`, которую я предоставил в исходном коде.

8.3.3. Каррирование и построение графика функции мгновенного расхода

Чтобы получить функцию, которая ведет себя подобно функции `flow_rate`, то есть принимает значение времени и возвращает значение расхода, нужно каррировать функцию `instantaneous_flow_rate`. Каррированная функция принимает функцию объема (`v`) и возвращает функцию расхода:

```
def get_flow_rate_function(v):
    def flow_rate_function(t):
        instantaneous_flow_rate(v,t)
    return flow_rate_function
```

`get_flow_rate_function(v)` возвращает другую функцию, идентичную функции `flow_rate` в примерах исходного кода. Чтобы подтвердить их идентичность, можно построить их графики на 10-часовом интервале. И действительно, как показывает рис. 8.17, их графики неразличимы:

```
plot_function(flow_rate,0,10)
plot_function(get_flow_rate_function(volume),0,10)
```

Мы решили первую основную задачу этой главы и создали функцию расхода на основе функции объема. Как упоминалось в начале главы, эта процедура называется *взятием производной*.

Для такой функции, как `volume`, другая функция, определяющая мгновенную скорость изменения объема в любой заданный момент времени, называется ее *производной*. Производную можно рассматривать как операцию, которая принимает одну (достаточно гладкую) функцию и возвращает другую функцию, оценивающую скорость изменения первой (рис. 8.18). В этом случае правильно было бы сказать, что функция расхода является производной функции объема.

Производная — это обобщенная процедура, применимая к *любой* функции $f(x)$, достаточно гладкой, чтобы иметь касательные прямые в каждой точке.

Производная функции f записывается как f' (произносится «f штрих»), соответственно, $f'(x)$ обозначает мгновенную скорость изменения f в зависимости от x. В частности, $f'(5)$ — это производная от $f(x)$ в точке с $x = 5$, она дает величину наклона касательной к f в точке $x = 5$. Существуют также некоторые другие распространенные обозначения производной функции, включая

$$f'(x) = \frac{df}{dx} = \frac{d}{dx} f(x).$$

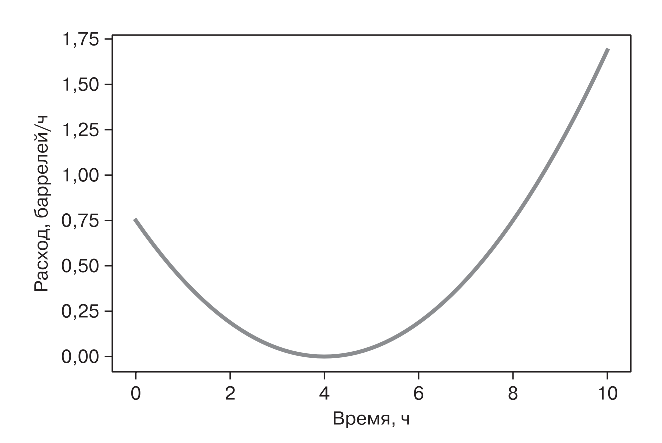


Рис. 8.17. Графики функций flow_rate и get_flow_rate неразличимы



Рис. 8.18. Производную можно рассматривать как некую машину, которая получает на входе одну функцию и возвращает другую, оценивающую скорость изменения функции на входе

Члены df и dx обозначают бесконечно малые изменения f и x соответственно, а их частное дает наклон бесконечно малой секущей прямой. Последняя форма записи с тремя членами хороша тем, что делает деление d/dx похожим на операцию, применяемую к $f(x)$. Отдельный оператор d/dx можно увидеть во многих контекстах. Он, в частности, означает операцию взятия производной по x. На рис. 8.19 схематично показано, как эти обозначения сочетаются друг с другом.

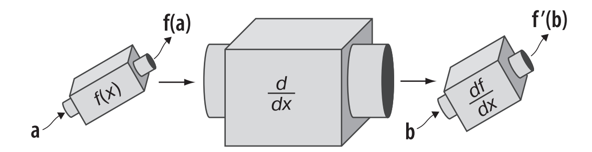


Рис. 8.19. Производная по *x* как операция, которая принимает одну функцию и возвращает другую

В оставшейся части этой книги мы часто будем использовать производные, а сейчас обратимся к противоположной операции — интегралу.

8.3.4. Упражнения

Упражнение 8.6. Покажите, что график функции `volume` — это не прямая линия на интервале от 0,999 до 1,001 часа.

Решение. Если бы это была прямая линия, то она совпадала бы со своей секущей в каждой точке. Однако секущая прямая на интервале 0,999–1,001 имеет на отметке $t = 1$ час другое значение, отличное от того, что дает функция `volume`:

```
>>> volume(1)
2.878125
>>> secant_line(volume,0.999,1.001)(1)
2.8781248593749997
```

Упражнение 8.7. Аппроксимируйте наклон касательной к графику изменения объема в точке $t = 8$, вычислив наклоны все меньших и меньших секущих вокруг точки $t = 8$.

Решение

```
>>> average_flow_rate(volume,7.9,8.1)
0.7501562500000007
>>> average_flow_rate(volume,7.99,8.01)
0.750001562499996
>>> average_flow_rate(volume,7.999,8.001)
0.7500000156249458
>>> average_flow_rate(volume,7.9999,8.0001)
0.7500000001554312
```

Судя по всему, мгновенная скорость изменения объема в точке $t = 8$ составляет 0,75 барреля в час.

Упражнение 8.8. Убедитесь, что функция `sign` не имеет производной в точке $x = 0$:

```
def sign(x):
    return x / abs(x)
```

Решение. На все меньших и меньших интервалах наклон секущей становится все больше и больше и не сходится к одному числу:

```
>>> average_flow_rate(sign,-0.1,0.1)
10.0
>>> average_flow_rate(sign,-0.01,0.01)
100.0
>>> average_flow_rate(sign,-0.001,0.001)
1000.0
>>> average_flow_rate(sign,-0.000001,0.000001)
1000000.0
```

Это связано с тем, что в точке $x = 0$ функция `sign` сразу же переходит от значения -1 к значению 1 и при увеличении ее график не выглядит как прямая линия.

8.4. АППРОКСИМАЦИЯ ИЗМЕНЕНИЯ ОБЪЕМА

В оставшейся части главы мы сосредоточимся на второй основной цели — восстановлении функции объема по известной функции расхода. Это процесс, обратный взятию производной, потому что заключается в восстановлении исходной функции по известной функции скорости ее изменения. В математическом анализе этот процесс называется *интегрированием*.

Я разобью задачу восстановления функции объема на несколько небольших примеров, которые помогут вам понять, как работает интегрирование. В первом примере мы напишем две функции, которые помогут определить, насколько изменился объем нефти в резервуаре за указанный период.

Назовем первую функцию `small_volume_change(q,t,dt)`. Она принимает время `t`, продолжительность интервала `dt` и функцию расхода `q`, которая возвращает приблизительное изменение объема между отметками времени t и $t + dt$. Функция расхода вычисляет результат, предполагая, что временной интервал слишком мал, чтобы на его протяжении расход мог значительно измениться.

Вторую функцию мы назовем `volume_change(q,t1,t2,dt)` и, как подсказывает разница в названиях, будем использовать ее для вычисления изменения объема не только на коротком, но и на любом временном интервале. Она будет принимать

функцию расхода q, время начала t1, время конца t2 и продолжительность короткого интервала времени dt. Функция volume_change будет разбивать большой временной интервал на приращения dt, достаточно маленькие для того, чтобы использовать функцию small_volume_change. Вычисленное общее изменение объема будет определяться как сумма всех изменений объема на коротких интервалах времени.

8.4.1. Вычисление изменения объема за короткий промежуток времени

Чтобы понять смысл функции small_volume_change, вернемся к примеру со спидометром в автомобиле. Если спидометр показывает скорость движения 60 миль/ч, то можно предположить, что в следующие 2 часа вы проедете 120 миль, умножив 2 часа на 60 миль/ч. Эта оценка может быть верной, если вам повезет, но также может случиться, что на пути вам встретится знак ограничения скорости или вы съедете с автострады и припаркуете машину. Дело в том, что одного взгляда на спидометр недостаточно, чтобы оценить расстояние, которое будет преодолено за длительный период времени.

С другой стороны, если после взгляда на спидометр использовать значение 60 миль/ч для оценки расстояния, которое будет пройдено в следующую секунду, то ответ наверняка получится очень точным; скорость движения не сильно изменится за одну секунду. Секунда — это 1/3600 часа, поэтому 60 миль/ч, умноженные на 1/3600 долю часа, дают 1/60 мили, или 88 футов (примерно 27 м). Если при этом вы не нажимаете энергично педаль тормоза или газа, то эта оценка будет весьма точной.

Теперь вернемся к расходу и объему и предположим, что мы используем довольно короткий интервал, в течение которого расход остается примерно постоянным. Иначе говоря, расход на временном интервале близок к среднему расходу на этом интервале, поэтому мы можем применить исходное уравнение:

$$\textbf{Расход} \approx \textbf{Средний расход} = \frac{\textbf{Изменение объема}}{\textbf{Прошедшее время}}.$$

Преобразовав его, получим формулу вычисления величины изменения объема:

$$\textbf{Изменение объема} \approx \textbf{Расход} \cdot \textbf{Прошедшее время}.$$

Функция small_volume_change — это просто перевод данной предполагаемой формулы на язык Python. Имея функцию расхода q, мы можем вычислить расход в момент времени t как q(t), умножить его на продолжительность dt и получить величину изменения объема:

```
def small_volume_change(q,t,dt):
    return q(t) * dt
```

У нас есть пара действующих функций объема и расхода, и теперь мы можем проверить, насколько хороша наша аппроксимация. Как и ожидалось, аппроксимация на часовом интервале оказалась далека от идеала:

```
>>> small_volume_change(flow_rate,2,1)
0.1875
>>> volume(3) - volume(2)
0.109375
```

Ошибка составила около 70 %. Для сравнения: на коротком временном интервале 0,01 часа оценка намного точнее. Ошибка не превышает 1 % от фактического изменения объема:

```
>>> small_volume_change(flow_rate,2,0.01)
0.001875
>>> volume(2.01) - volume(2)
0.0018656406250001645
```

Поскольку мы можем получить хорошие аппроксимации изменения объема на коротких временных интервалах, можно попробовать собрать их вместе, чтобы получить изменение объема на более длинном интервале.

8.4.2. Разбиение временного отрезка на мелкие интервалы

Чтобы реализовать функцию `volume_change(q, t1, t2, dt)`, нужно разбить весь временной отрезок от `t1` до `t2` на интервалы длительностью `dt`. Для простоты будем использовать только значения `dt`, которые делят `t2 - t1` на целое число интервалов.

И снова для получения времени начала каждого интервала можно применить функцию `arange` из библиотеки NumPy. Вызов функции `np.arange(t1, t2, dt)` даст нам массив интервалов от `t1` до `t2` длительностью `dt` каждый. Для каждого значения времени `t` в этом массиве можно найти величину изменения объема на соответствующем временном интервале с помощью `small_volume_change` и затем сложить результаты, чтобы получить общее изменение объема на всем временном отрезке. Все это можно выразить одной строкой кода:

```
def volume_change(q,t1,t2,dt):
    return sum(small_volume_change(q,t,dt)
               for t in np.arange(t1,t2,dt))
```

С помощью этой функции мы можем разбить временной отрезок от 0 до 10 часов на 100 интервалов продолжительностью 0,1 часа и сложить изменения объема на каждом из них. Как можно видеть далее, результат соответствует фактическому изменению объема с точностью до одного знака после запятой:

```
>>> volume_change(flow_rate,0,10,0.1)
4.32890625
>>> volume(10) - volume(0)
4.375
```

Если разбивать временной отрезок на все меньшие интервалы, то результат будет последовательно улучшаться, например:

```
>>> volume_change(flow_rate,0,10,0.0001)
4.3749531257812455
```

Так же как в случае с производной, интервалы можно уменьшать все больше и больше, и результаты будут сходиться к ожидаемому ответу. Вычисление общего изменения функции на некотором интервале по скорости ее изменения называется *определенным интегралом*. Мы вернемся к определению такого интеграла в последнем разделе этой главы, а пока посмотрим, как его изобразить.

8.4.3. Изображение изменения объема на графике расхода

Предположим, что мы разбили 10-часовой временной отрезок на часовые интервалы, пусть даже знаем, что это не даст точных результатов. На графике расхода нас интересуют только 10 точек, соответствующих времени начала каждого интервала: 0 часов, 1 час, 2 часа, 3 часа и т. д. вплоть до 9 часов. Эти точки показаны на графике на рис. 8.20.

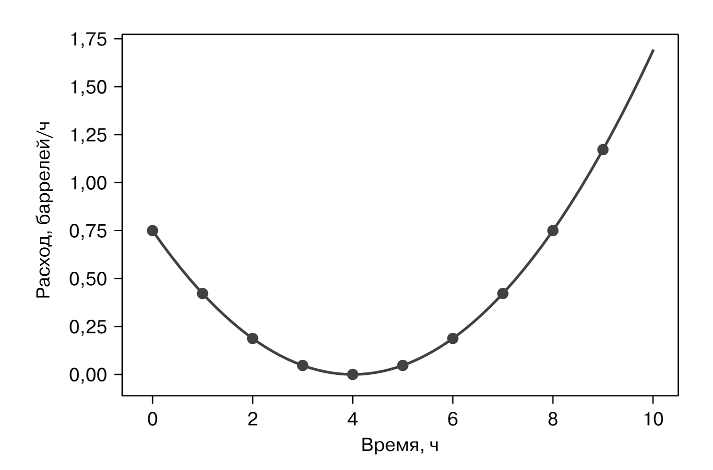


Рис. 8.20. Изображение точек, используемых в вычислениях с помощью volume_change(flow_rate, 0,10,1)

В своих вычислениях мы предполагали, что расход в пределах каждого интервала остается постоянным, а это явно не соответствует действительности. На каждом из этих интервалов он заметно меняется. Сделав предположение, мы как бы взяли за основу другую функцию расхода, график которой постоянен

в течение каждого часа. На рис. 8.21 показано, как он выглядит на фоне графика оригинальной функции.

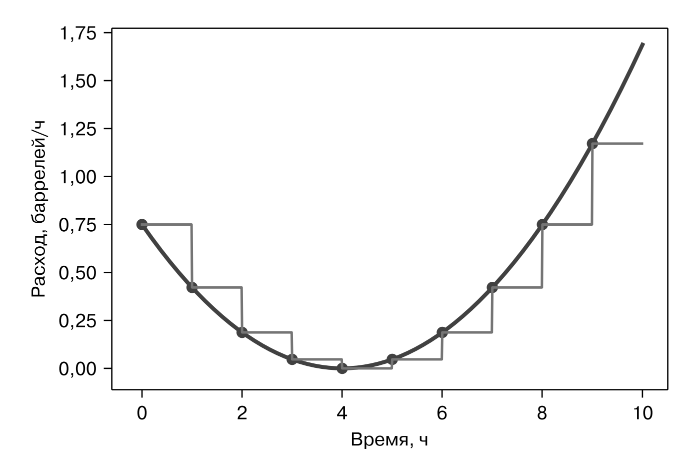


Рис. 8.21. Если предположить, что расход постоянен на каждом интервале, то график функции выглядел бы как лестница, по которой можно спуститься и снова подняться

На каждом интервале мы вычисляем расход (определяется как высота каждого столбика на графике) и умножаем его на длительность интервала, равную 1 часу (ширина каждого столбика). Проще говоря, изменение объема на каждом коротком интервале вычисляется как произведение высоты на ширину соответствующего столбика на графике, то есть как площадь воображаемого прямоугольника. Для большей ясности эти прямоугольники показаны на рис. 8.22.

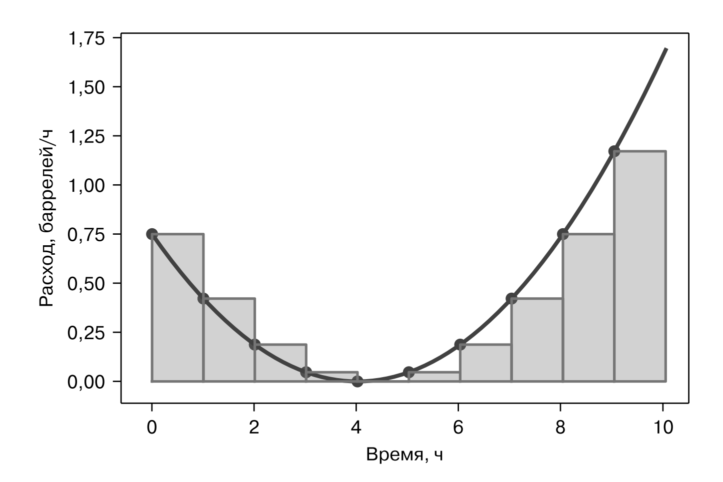


Рис. 8.22. Общее изменение объема равно сумме площадей 10 прямоугольников

Чем меньше интервалы, тем лучше получаются результаты. Визуально это соответствует большему количеству прямоугольников, полнее охватывающих график. На рис. 8.23 показано, как выглядят прямоугольники при разбиении на 30 интервалов по 1/3 часа (20 минут) и при разбиении на 100 интервалов по 0,1 часа.

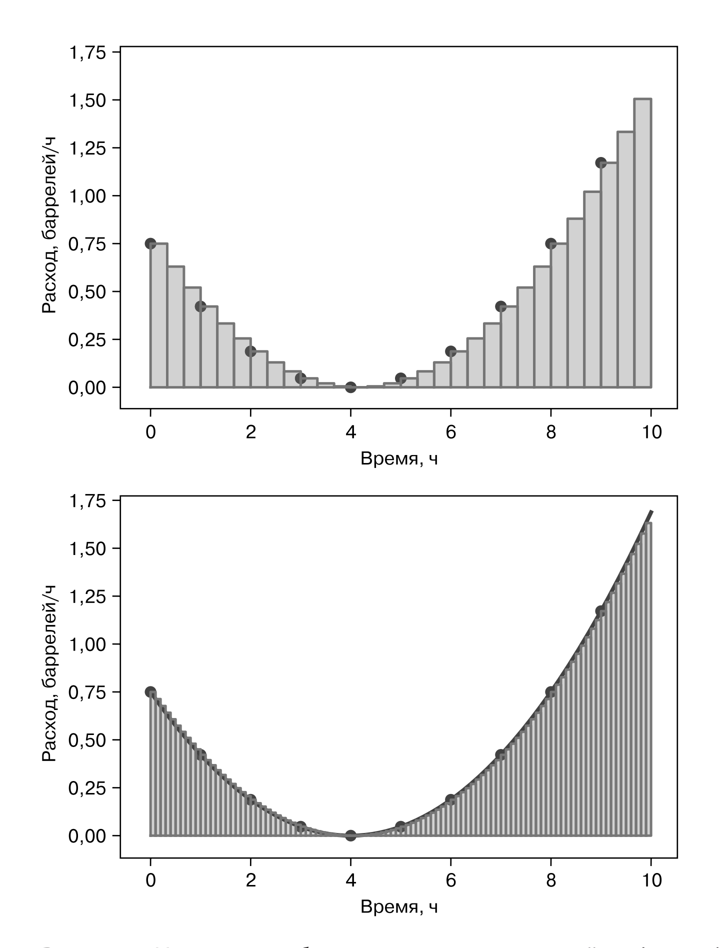


Рис. 8.23. Изменение объема как сумма площадей 30 *(вверху)* или 100 прямоугольников *(внизу)* под графиком расхода (см. рис. 8.20)

На этих графиках ясно видно, что с уменьшением интервалов вычисленный результат приближается к фактическому изменению объема — прямоугольники все полнее и полнее покрывают пространство под графиком расхода. Здесь важно понять, что площадь под графиком расхода на заданном временном отрезке *(приблизительно) равна* объему нефти, добавленной в резервуар за то же время.

Сумма площадей прямоугольников, аппроксимирующая площадь под графиком, называется *суммой Римана*. Суммы Римана, составленные из все более и более узких прямоугольников, сходятся к площади под графиком почти так же, как наклоны все более коротких секущих сходятся к наклону касательной. Мы еще вернемся к сходимости сумм Римана и определенных интегралов, а пока продолжим решать задачу нахождения изменения объема во времени.

8.4.4. Упражнения

Упражнение 8.9. Сколько приблизительно нефти было добавлено в резервуар за первые 6 часов? За последние 4 часа? На каком промежутке времени было добавлено больше нефти?

Решение. За первые 6 часов в резервуар было закачано около 1,13 барреля нефти, что меньше, чем примерно 3,24 барреля, закачанных в резервуар за последние 4 часа:

```
>>> volume_change(flow_rate,0,6,0.01)
1.1278171874999996
>>> volume_change(flow_rate,6,10,0.01)
3.2425031249999257
```

8.5. ГРАФИК ИЗМЕНЕНИЯ ОБЪЕМА С ТЕЧЕНИЕМ ВРЕМЕНИ

В предыдущем разделе у нас вышло, начав с расхода, получить приблизительные значения *изменения* объема за заданный интервал времени. Главная же цель — вычислить *общий* объем нефти в резервуаре в любой момент времени.

Попробуйте ответить на каверзный вопрос: если в течение 3 часов нефть поступает в резервуар с постоянной скоростью 1,2 барреля в час, то сколько нефти будет в нем через 3 часа? Ответ: мы не знаем, потому что в условиях задачи не говорится, сколько нефти было в резервуаре изначально! Но если я назову это число, то вы легко найдете ответ. Например, если вначале в резервуаре имелось 0,5 барреля, то за указанный период в него поступит 3,6 барреля и общий объем составит 0,5 + 3,6 = 4,1 барреля. Прибавив начальный объем в нулевой момент времени к изменению объема в любой момент времени T, можно найти общий объем в момент времени T.

В примерах в данном разделе мы превратим эти рассуждения в код и восстановим функцию объема. Мы реализуем функцию `approximate_volume(q, v0, dt, T)`,

которая принимает функцию расхода q, начальный объем нефти в резервуаре v0, продолжительность короткого интервала времени dt и интересующее нас время T. Результатом функции будет примерное значение общего объема нефти в резервуаре в момент времени T, вычисляемого как сумма начального объема v0 и изменения объема от нулевой отметки времени до отметки T.

Затем каррируем эту функцию, чтобы получить функцию approximate_volume_function(q, v0, dt), которая возвращает функцию, вычисляющую приблизительный объем как функцию от времени. Функция, возвращаемая функцией approximate_volume_function, — это функция объема, и мы сможем построить ее график на фоне графика исходной функции объема для сравнения.

8.5.1. Вычисление объема в заданный момент времени

Вот как выглядит базовая формула, которую мы будем использовать:

$$\textbf{объем в момент времени } T =$$
$$= (\textbf{объем в момент времени } 0) + (\textbf{изменение объема за период от } 0 \textbf{ до } T).$$

Мы должны явно задать первое слагаемое суммы — объем нефти в резервуаре в нулевой момент времени, потому что вывести это число из функции расхода невозможно. Затем можно использовать функцию volume_change, чтобы найти изменение объема от нулевого времени до времени T. Вот как выглядит реализация:

```
def approximate_volume(q,v0,dt,T):
    return v0 + volume_change(q,0,T,dt)
```

Чтобы каррировать эту функцию, определим новую функцию, принимающую первые три аргумента в виде параметров и возвращающую новую функцию, которая принимает последний параметр T:

```
def approximate_volume_function(q,v0,dt):
    def volume_function(T):
        return approximate_volume(q,v0,dt,T)
    return volume_function
```

Следующая функция рисует график функции объема, используя функцию flow_rate. Поскольку функция volume, которую вы найдете в примерах исходного кода для книги, возвращает значение 2,3 для $T = 0$, передадим это значение в параметре v0. Наконец, попробуем для начала значение dt, равное 0,5, то есть будем рассчитывать изменения объема с получасовыми (30-минутными) интервалами. Посмотрим, как выглядит график полученной нами функции на фоне исходной функции объема (рис. 8.24):

```
plot_function(approximate_volume_function(flow_rate,2.3,0.5),0,10)
plot_function(volume,0,10)
```

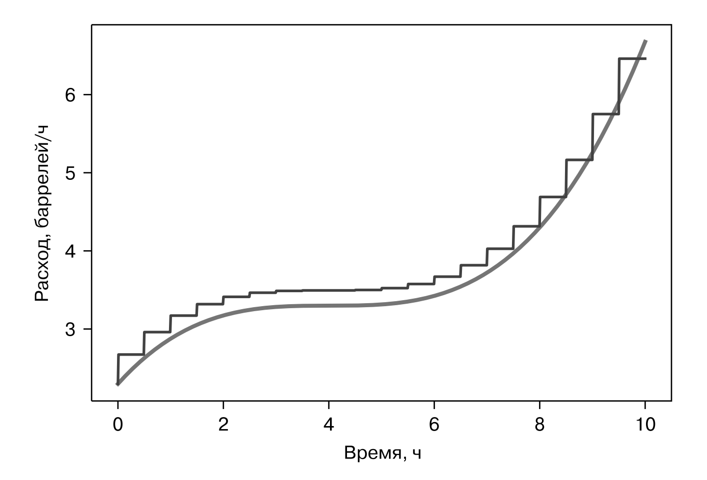


Рис. 8.24. График функции approximate_volume_function (*ступенчатая линия*) на фоне графика исходной функции объема (*гладкая кривая*)

Как видите, мы получили результат, довольно близкий к исходной функции объема! Но график `approximate_volume_function` выглядит как ступенчатая линия с шагом 0,5 часа. Нетрудно догадаться, что это связано со значением `dt`, равным 0,5, и что можно получить лучшее приближение, если уменьшить его. Это верная догадка, но рассмотрим подробнее, как вычисляется изменение объема, чтобы точно понять, почему график выглядит именно так и почему меньший временной интервал улучшит его.

8.5.2. Представление сумм Римана для функции объема

Объем нефти в резервуаре в любой момент времени, полученный с помощью функции `approximate_volume_function`, вычисляется как сумма начального объема нефти в резервуаре и его изменения к заданному моменту времени. Для $t = 4$ уравнение выглядит так:

$$\textbf{объем в момент времени } 4 =$$
$$= \left(\textbf{объем в момент времени } 0\right) + \left(\textbf{изменение объема за период от } 0 \textbf{ до } 4\right).$$

Эта сумма дает точку на графике на 4-часовой отметке. Значение, соответствующее любому другому моменту времени, вычисляется точно так же. В данном случае слагаемыми являются 2,3 барреля в нулевое время и сумма Римана, дающая изменение объема в период от 0 часов до 4 часов. Сумма Римана складывается из площадей восьми прямоугольников, каждый из которых имеет ширину 0,5 часа, равномерно вписанных в четырехчасовое окно. В результате получается примерно 3,5 барреля (рис. 8.25).

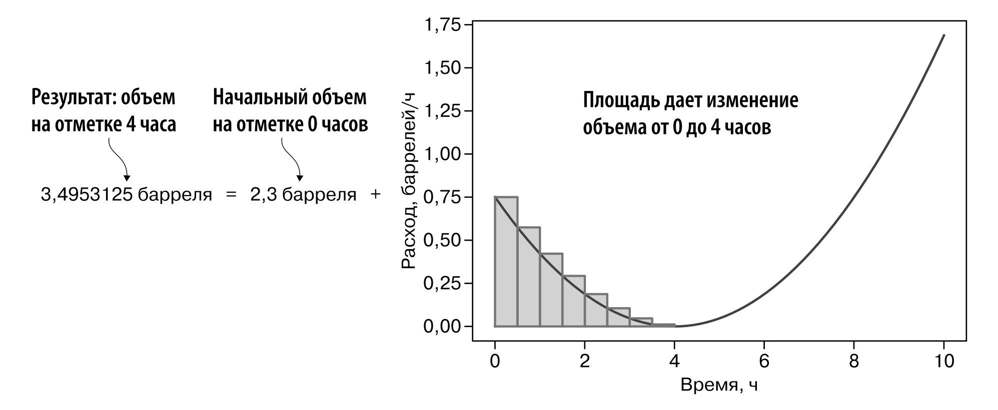


Рис. 8.25. Объем нефти в резервуаре через 4 часа, полученный вычислением суммы Римана

Точно такие же вычисления можно выполнить для любого другого момента времени. Например, на рис. 8.26 показан результат для отметки 8 часов.

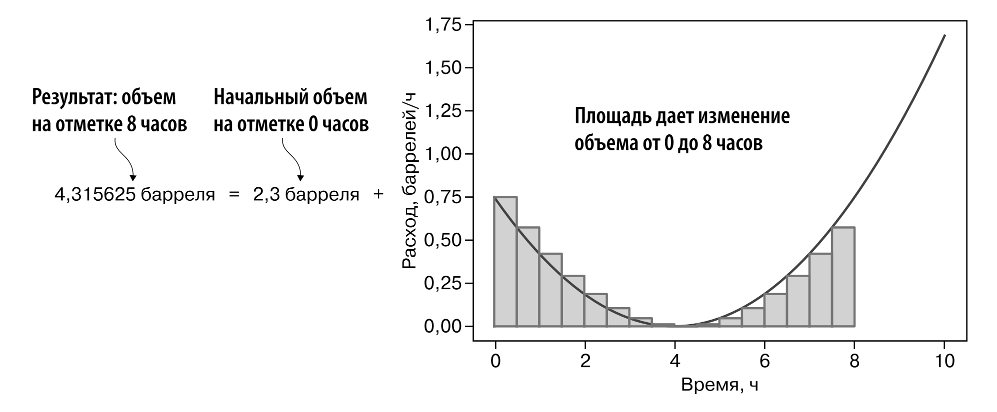


Рис. 8.26. Объем нефти в резервуаре через 8 часов, полученный вычислением суммы Римана

Ответ в этом случае: на восьмичасовой отметке времени в резервуаре находится примерно 4,32 барреля нефти. Чтобы найти его, потребовалось сложить площади 8/0,5 = 16 прямоугольников. Два найденных нами ответа показаны точками на графике (рис. 8.27).

В обоих случаях мы добрались от нуля до рассматриваемого момента времени, используя целое количество временных шагов. Чтобы создать этот график, код на Python вычислил множество сумм Римана, соответствующих целому числу часов и получасов, а также всем точкам, нанесенным на график между ними.

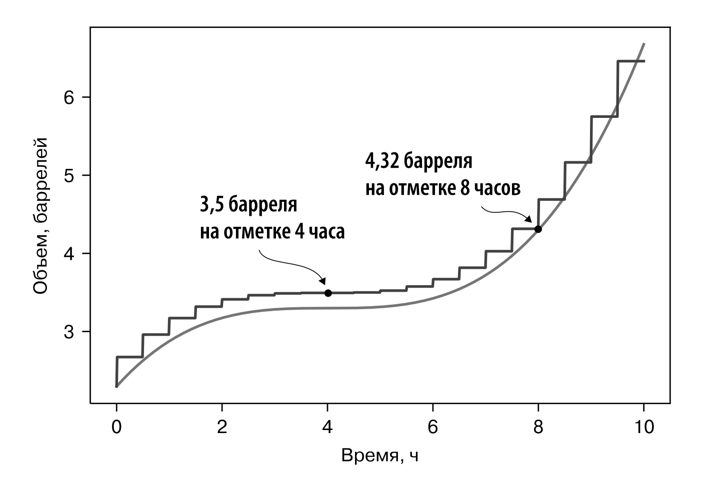


Рис. 8.27. Два приблизительных результата показаны на графике изменения объема

Но сможет ли наш код получить приблизительный объем нефти на отметке 3,9 часа, который не делится без остатка на значение *dt*, равное 0,5 часа? Вернемся к реализации `volume_change(q,t1,t2,dt)`, вычисляющей изменение объема, которое соответствует площади одного прямоугольника из числа возвращаемых вызовом `np.arange(t1,t2,dt)`. Если попытаться найти изменение объема на отрезке от 0 до 3,9 часа с *dt* = 0,5, то `np.arange` вернет список прямоугольников:

```
>>> np.arange(0,3.9,0.5)
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5])
```

Несмотря на то что восемь прямоугольников шириной по 0,5 часа выходят за отметку времени 3,9 часа, функция вычислит площадь всех восьми полных прямоугольников! Чтобы добиться большей точности, мы, вероятно, должны сократить временной интервал *dt* до 0,4 часа после обработки 7-го временного интервала, заканчивающегося на отметке 3,5 часа, чтобы не уйти дальше конечной отметки 3,9 часа. Попробуйте в качестве самостоятельного упражнения изменить функцию `volume_change` так, чтобы она использовала меньшую продолжительность для последнего временного интервала, если это необходимо. А я просто проигнорирую эту оплошность.

В предыдущем разделе мы видели, что более точный результат можно получить, уменьшив значение *dt* и, следовательно, ширину прямоугольников. Кроме более полного охвата площади под линией графика меньшие прямоугольники, вероятно, дадут меньшую ошибку, даже если будут немного выступать за конец временного отрезка. Например, 0,5-часовые интервалы равномерно заполняют отрезок времени 3,5 или 4 часа, но не 3,9 часа, а 0,1-часовые интервалы могут равномерно заполнить и отрезок 3,9 часа.

8.5.3. Улучшение аппроксимации

Попробуем использовать меньшие значения *dt*, соответствующие меньшим размерам прямоугольника, и посмотрим, насколько улучшится точность. Вот аппроксимация с `dt` = 0,1 часа (результаты показаны на рис. 8.28). Ступени на графике едва видны, они стали меньше, и график аппроксимации намного ближе к реальному графику функции изменения объема, чем при 0,5-часовых интервалах:

```
plot_function(approximate_volume_function(flow_rate,2.3,0.1),0,10)
plot_function(volume,0,10)
```

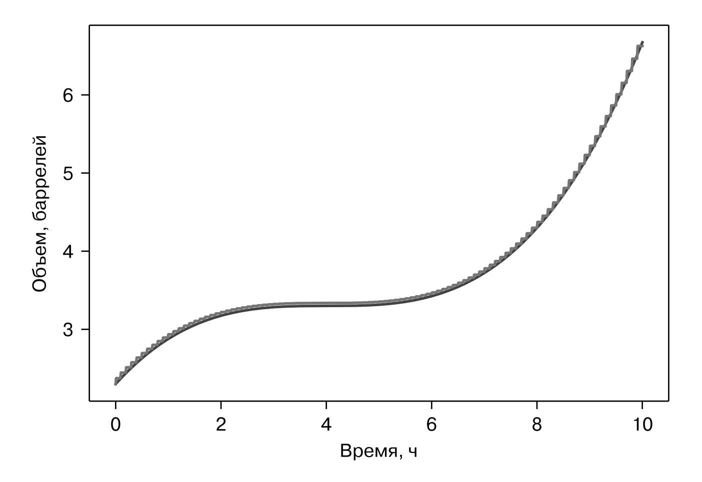


Рис. 8.28. При *dt* = 0,1 часа графики почти совпадают

Если величину временных шагов еще уменьшить, например до *dt* = 0,01 часа, то графики становятся практически неразличимы (рис. 8.29):

```
plot_function(approximate_volume_function(flow_rate,2.3,0.01),0,10)
plot_function(volume,0,10)
```

Несмотря на кажущееся совпадение графиков, у нас все же может возникнуть вопрос: насколько точна эта аппроксимация? Графики аппроксимаций функции `volume` со все меньшими и меньшими значениями `dt` становятся все ближе и ближе к фактическому графику изменения объема в каждой его точке, поэтому можно сказать, что значения аппроксимаций *сходятся* к фактическим значениям объема. Но на каждом конкретном шаге аппроксимация все еще может не совпадать с фактическим значением объема.

Мы могли бы рассчитать объем в любой точке с произвольной точностью (в пределах выбранного допуска), придерживаясь следующего алгоритма: для любого момента времени `t` вычислять `volume_change(q, 0, t, dt)` со все уменьшающимися значениями `dt`, пока результаты не перестанут изменяться более чем

на величину допуска. Это очень похоже на то, как мы многократно вычисляли производную функции, пока она не стабилизируется:

```
def get_volume_function(q,v0,digits=6):
    def volume_function(T):
        tolerance = 10 ** (-digits)
        dt = 1
        approx = v0 + volume_change(q,0,T,dt)
        for i in range(0,digits*2):
            dt = dt / 10
            next_approx = v0 + volume_change(q,0,T,dt)
            if abs(next_approx - approx) < tolerance:
                return round(next_approx,digits)
            else:
                approx = next_approx
        raise Exception("Did not converge!")
    return volume_function
```

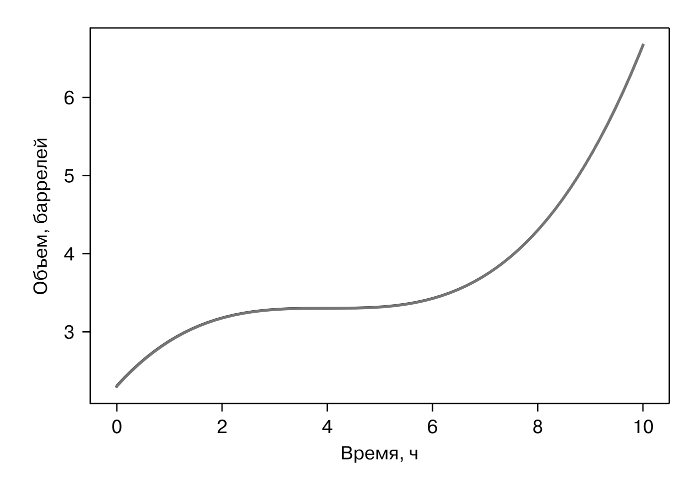


Рис. 8.29. При *dt* = 0,01 часа график аппроксимации неотличим от фактической функции изменения объема

В частности, точное значение объема $v(1)$ составляет 2,878125 барреля, и мы можем аппроксимировать объем в заданной точке с любой точностью, например, с точностью до трех знаков после запятой:

```
>>> v = get_volume_function(flow_rate,2.3,digits=3)
>>> v(1)
2.878
```

или с точностью до шести знаков:

```
>>> v = get_volume_function(flow_rate,2.3,digits=6)
>>> v(1)
2.878125
```

Попробовав запустить этот код у себя, вы заметите, что второе вычисление занимает довольно много времени. Это связано с тем, что для получения ответа с такой точностью необходимо вычислить сумму Римана, состоящую из миллионов небольших изменений объема. Возможно, функция, позволяющая вычислить аппроксимацию с произвольной точностью, не найдет реального применения, но она иллюстрирует тот факт, что с уменьшением значения dt результаты вычислений сходятся к точному значению функции объема. Значение, к которому сходятся результаты, называется *интегралом* расхода.

8.5.4. Определенные и неопределенные интегралы

В последних двух разделах мы *интегрировали* функцию расхода, чтобы получить функцию объема. Подобно нахождению производной, вычисление интеграла — это обобщенная процедура, которую можно применять к функциям. Мы можем интегрировать любую функцию, определяющую скорость изменения, чтобы получить функцию, дающую совместимое накопленное значение. Если, например, скорость автомобиля известна как функция от времени, то мы можем интегрировать ее, чтобы получить пройденное расстояние как функцию от времени. В этом разделе рассмотрим два типа интегралов — определенные и неопределенные.

Определенный интеграл сообщает общее изменение функции на некотором интервале по ее производной. Функция вместе с начальным и конечным значениями аргумента, которым в нашем случае является время, задают определенный интеграл. Результатом становится одно число, представляющее накопленное изменение. Например, если $f(x)$ — интересующая нас функция, а $f'(x)$ — производная функции $f(x)$, то изменение f на интервале от $x = a$ до $x = b$ будет равно разности $f(b) - f(a)$, которую можно найти, взяв определенный интеграл (рис. 8.30).

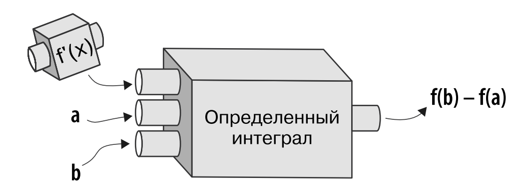


Рис. 8.30. Определенный интеграл получает скорость изменения (производную) функции и заданный интервал и восстанавливает накопленное изменение функции на этом интервале

В матанализе определенный интеграл $f(x)$ на интервале от $x = a$ до $x = b$ записывается так:

$$\int_a^b f'(x)dx,$$

а его значение равно разности $f(b) - f(a)$. Большой символ $\int$ — это символ интеграла, a и b называются *границами интегрирования*, $f'(x)$ — интегрируемая функция, а dx указывает, что интеграл берется по x.

Функция `volume_change` аппроксимирует определенные интегралы и, как мы видели в разделе 8.4.3, аппроксимирует также площадь под графиком расхода. Оказывается, определенный интеграл функции на интервале равен площади под графиком на этом интервале. Большинство функций, встречающихся на практике, имеют графики достаточно гладкие для того, чтобы площадь под ними можно было аппроксимировать с помощью все более и более узких прямоугольников, и для них аппроксимации будут сходиться к одному значению.

Теперь посмотрим на неопределенный интеграл. *Неопределенный интеграл* принимает производную функции и восстанавливает исходную функцию. Например, если известно, что $f'(x)$ — это производная от $f(x)$, то для восстановления $f(x)$ нужно найти неопределенный интеграл от $f'(x)$.

Загвоздка в том, что одной производной $f'(x)$ недостаточно для восстановления исходной функции $f(x)$. Как мы видели на примере функции `get_volume_function`, которая вычисляет определенный интеграл, необходимо знать начальное значение $f(x)$, например $f(0)$. Тогда значение $f(x)$ можно найти, прибавив определенный интеграл к $f(0)$. Поскольку

$$f(b) - f(a) = \int_a^b f'(x)dx,$$

можно получить любое значение $f(x)$ как:

$$f(x) - f(0) = \int_0^x f'(t)dt.$$

Обратите внимание на то, что мы должны использовать другое имя t для аргумента f, потому что здесь x становится границей интегрирования. Неопределенный интеграл от функции $f(x)$ записывается как

$$f(x) = \int f'(x)dx$$

и выглядит так же, как определенный интеграл, но без границ. Если, например, $g(x) = \int f(x)dx$, то говорят, что $g(x)$ — это первообразная (antiderivative) функции $f(x)$. Первообразные не уникальны, и на самом деле существует другая функция $g(x)$, производная которой равна $f(x)$ для любого выбранного начального значения $g(0)$.

Мы познакомились с большим количеством новых терминов, но, к счастью, остаток части II книги посвятим их практическому обзору. Продолжим работать с функциями и скоростью их изменения, используя производные и интегралы для взаимозаменяемого переключения между ними.

КРАТКИЕ ИТОГИ ГЛАВЫ

- Средняя скорость изменения функции, скажем $f(x)$, есть изменение значения f на некотором интервале x, деленное на длину интервала. Например, средняя скорость изменения $f(x)$ на интервале от $x = a$ до $x = b$ равна

$$\frac{f(b)-f(a)}{b-a}.$$

- Среднюю скорость изменения функции можно изобразить как *секущую прямую*, проходящую через график функции в двух точках.
- График гладкой функции при увеличении выглядит неотличимым от прямой линии. Прямая, на которую похожа линия графика, — это наилучшая линейная аппроксимация функции в этой области, а ее наклон называется *производной* функции.
- Вы можете аппроксимировать производную, определяя наклоны секущих на все уменьшающихся интервалах в окрестностях заданной точки. Полученное значение аппроксимирует мгновенную скорость изменения функции в этой точке.
- *Производная* функции — это еще одна функция, которая дает мгновенную скорость изменения в каждой точке. Вы можете построить график производной функции, чтобы увидеть, как меняется скорость ее изменения с течением времени.
- Имея производную функции, можно выяснить, как она изменяется во времени, разбив отрезок времени на короткие интервалы и предположив, что на каждом из них скорость постоянна. Если интервалы достаточно короткие, то вычисленная скорость будет приблизительно постоянной, а суммированием можно найти общее количество. Это значение аппроксимирует определенный интеграл функции.
- Зная начальное значение функции и взяв определенный интеграл от ее скорости на различных интервалах, можно восстановить функцию. Такой интеграл называется *неопределенным интегралом* функции.

9

Моделирование перемещающихся объектов

В этой главе

- ✓ Реализация законов движения Ньютона для реалистичной имитации движения.
- ✓ Вычисление векторов скорости и ускорения.
- ✓ Использование метода Эйлера для аппроксимации положения движущегося объекта.
- ✓ Определение точной траектории движения объекта с помощью математического анализа.

Игра «Астероиды» из главы 7 — вполне действующая, но не особенно сложная. Чтобы было интереснее в нее играть, нужно, чтобы астероиды двигались! А чтобы дать игроку возможность уклоняться от движущихся астероидов, следует сделать движущимся космический корабль и позволить управлять им.

Для реализации движения в игре с астероидами воспользуемся положениями из матанализа, представленными в главе 8. Числовые величины, которые мы будем рассматривать, — это координаты x и y астероидов и космического корабля. Чтобы астероиды двигались, эти величины должны меняться с течением времени, поэтому будем считать их функциями от времени $x(t)$ и $y(t)$. Производная функции местоположения по времени называется *скоростью*, а производная скорости по времени — *ускорением*. У нас есть две функции местоположения, соответственно, должны быть две функции скорости и две функции ускорения. Это позволяет рассматривать скорости и ускорения как векторы.

Наша первая цель — реализовать движение астероидов. Для этого определим функции, возвращающие случайные и постоянные скорости для астероидов. Затем будем интегрировать эти функции скорости в реальном времени, чтобы определить местоположение каждого астероида в каждом кадре, используя алгоритм, называемый *методом Эйлера*. Математически метод Эйлера подобен интегрированию, рассмотренному в главе 8, но у него есть одно преимущество — его можно применять по ходу игры.

Далее реализуем возможность управления космическим кораблем. При нажатии клавиши со стрелкой вверх космический корабль будет ускоряться в том направлении, куда он смотрит. То есть производная от производной каждой из функций $x(t)$ и $y(t)$ станет отличной от нуля: скорость начинает меняться, и вместе с ней начнет меняться местоположение. И снова для интегрирования функций ускорения и скорости в реальном времени используем метод Эйлера.

Метод Эйлера — это всего лишь аппроксимация интеграла, и с этой точки зрения он подобен суммам Римана, о которых рассказывалось в главе 8. С его помощью можно вычислить точное местоположение астероидов и космического корабля во времени. В заключение главы я дам краткое сравнение результатов метода Эйлера и точных решений.

9.1. ИМИТАЦИЯ ДВИЖЕНИЯ С ПОСТОЯННОЙ СКОРОСТЬЮ

В повседневном обиходе слово *«скорость»* означает просто *скорость движения*. В математике и физике понятие скорости включает также направление. Поэтому мы сосредоточимся на математическом понятии скорости и будем рассматривать ее как вектор.

Наша первоочередная цель — придать каждому из астероидов случайный вектор скорости, то есть пару чисел (v_x, v_y), и интерпретировать их как постоянные значения производных местоположения по времени. То есть будем предполагать, что $x'(t) = v_x$ и $y'(t) = v_y$. Реализовав представление этой информации, мы сможем обновить игровой движок и заставить астероиды двигаться с заданными скоростями.

Поскольку игра двухмерная, мы будем работать с парами координат и парами скоростей. В обсуждении я буду говорить об $x(t)$ и $y(t)$ как о паре функций местоположения, а об $x'(t)$ и $y'(t)$ как о паре функций скорости и стану записывать их как *векторные функции*: $\mathbf{s}(t) = (x(t), y(t))$ и $\mathbf{v}(t) = (x'(t), y'(t))$. Эти формы записи просто означают, что $\mathbf{s}(t)$ и $\mathbf{v}(t)$ являются функциями, которые принимают значение времени и возвращают вектор, представляющий местоположение и скорость в указанный момент времени.

У астероидов уже есть векторы местоположения, определяемые их свойствами x и y, но нам также нужно задать векторы скорости, указывающие, насколько быстро они движутся в направлениях x и y. Это первый шаг к поставленной цели — заставить астероиды перемещаться от кадра к кадру.

9.1.1. Добавление в астероиды информации о скоростях

Чтобы придать каждому астероиду вектор скорости, добавим в объект `PolygonModel` два компонента вектора v_x и v_y в форме свойств. Вот как это сделано в версии `asteroids.py` для главы 9 в примерах исходного кода:

```
class PolygonModel():
    def __init__(self,points):
        self.points = points
        self.angle = 0
        self.x = 0
        self.y = 0
        self.vx = 0
        self.vy = 0
```

Первые четыре свойства остались прежними, как в исходной реализации этого класса в главе 7

Свойства vx и vy предназначены для хранения текущих значений vx = x'(t) и vy = y'(t). По умолчанию им присваивается значение 0, то есть объект не движется

Чтобы астероиды двигались хаотично, можно задать им случайные значения скоростей. Для этого добавим две строки в конец конструктора `Asteroid`:

```
class Asteroid(PolygonModel):
    def __init__(self):
        sides = randint(5,9)
        vs = [vectors.to_cartesian((uniform(0.5,1.0), 2 * pi * i / sides))
              for i in range(0,sides)]
        super().__init__(vs)
        self.vx = uniform(-1,1)
        self.vy = uniform(-1,1)
```

До этой строки код не изменился по сравнению с главой 7, он инициализирует астероид в форме многоугольника со случайно расположенными вершинами

В последних двух строках задаются случайные значения скорости от –1 до 1 вдоль осей x и y

Напомню, что отрицательная производная говорит об убывании функции, а положительная — о возрастании. Положительные и отрицательные скорости x и y говорят об увеличении и уменьшении x и y соответственно. То есть астероиды могут двигаться вправо или влево, вверх или вниз.

9.1.2. Добавление поддержки перемещения астероидов в игровой движок

Далее мы должны использовать значения скоростей для обновления местоположения игровых объектов. Для любых объектов, будь то космический корабль или астероиды, компоненты скорости v_x и v_y сообщают, как следует изменить компоненты местоположения x и y.

Если между кадрами проходит некоторое время Δt, то координата x изменяется на величину $v_x \cdot \Delta t$, а координата y — на $v_y \cdot \Delta t$. (Символ Δ — это прописная греческая буква «дельта», она обычно применяется для обозначения приращения переменной.) Это та же самая аппроксимация, которую мы использовали, чтобы определить величину изменения объема по изменению расхода в главе 8. В данном случае произведение скорости на прошедшее время дает величину изменения позиции, и это не аппроксимация, а точная оценка, потому что скорости постоянны.

Мы можем добавить в класс `PolygonModel` метод перемещения `move`, обновляющий местоположение объекта на основе этой формулы. Единственное, что неизвестно объекту, — это прошедшее время, поэтому передаем его как аргумент (в миллисекундах):

```
class PolygonModel():
    ...
    def move(self, milliseconds):
            dx, dy = (self.vx * milliseconds / 1000.0,
                      self.vy * milliseconds / 1000.0
            self.x, self.y = vectors.add((self.x,self.y),
                                         (dx,dy))
```

Приращение координаты x называется dx, а приращение координаты y называется dy. Оба вычисляются как произведение скорости на прошедшее время в секундах

Завершает вычисление местоположения для данного кадра, прибавляя соответствующие приращения dx и dy к текущим координатам

Это первое применение нами метода Эйлера на практике. Суть алгоритма заключается в отслеживании значений одной или нескольких функций (в данном случае позиций $x(t)$ и $y(t)$, а также их производных $x'(t) = v_x$ и $y'(t) = v_y$) и их изменении на каждом шаге в соответствии со значениями производных. Этот метод дает точные результаты, когда производные постоянны, и неплохое приближение, когда они меняются. Перейдя к реализации перемещения космического корабля, мы вплотную столкнемся с изменением значений скорости и дополним нашу реализацию метода Эйлера.

9.1.3. Удержание астероидов в пределах экрана

Мы можем добавить еще одну небольшую деталь, чтобы усовершенствовать игровой процесс. Астероид, движущийся со случайной скоростью, в какой-то момент обязательно достигнет границы экрана. Чтобы астероиды не покидали игровой сцены, можно добавить дополнительную логику, обеспечивающую удержание обеих координат в границах между минимальным и максимальным значениями от –10 до 10. Когда, например, координата x со значением 10,0 увеличивается до 10,1, вычтем из нее 20, чтобы она получила допустимое значение –9,9. Это создаст эффект телепортации астероида из правой части экрана в левую. Такая

игровая механика не имеет ничего общего с физикой, но делает игру более интересной, не позволяя астероидам покинуть экран (рис. 9.1).

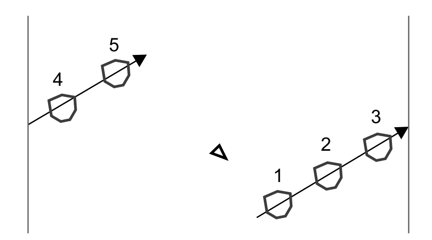


Рис. 9.1. Удержание координат всех объектов в диапазоне от –10 до 10 путем телепортации объектов от одного края экрана к другому при его пересечении

Вот код, создающий эффект телепортации:

```
class PolygonModel():
    ...
    def move(self, milliseconds):
        dx, dy = (self.vx * milliseconds / 1000.0,
                  self.vy * milliseconds / 1000.0)
        self.x, self.y = vectors.add((self.x,self.y),
                                      (dx,dy))
        if self.x < -10:
            self.x += 20
        if self.y < -10:
            self.y += 20
        if self.x > 10:
            self.x -= 20
        if self.y > 10:
            self.y -=20
```

Если x < –10, значит, астероид переместился за левую границу экрана, поэтому прибавляем 20 единиц к позиции x, чтобы телепортировать его в правую часть экрана

Если y < –10, значит, астероид переместился за нижнюю границу экрана, поэтому прибавляем 20 единиц к позиции y, чтобы телепортировать его в верхнюю часть экрана

Наконец, нам нужно вызвать метод `move` для каждого астероида в игре. Для этого добавим в игровой цикл перед началом рисования следующие строки:

```
milliseconds = clock.get_time()
for ast in asteroids:
    ast.move(milliseconds)
```

Вычислить количество миллисекунд, прошедших после рисования предыдущего кадра

Подать сигнал всем астероидам, что они должны обновить свое местоположение в зависимости от скорости

На картинке, изображенной на странице книги, этого не видно, но если вы запустите этот код у себя, то увидите, что астероиды беспорядочно перемещаются по экрану в случайных направлениях. Сосредоточив внимание на каком-то одном астероиде, вы заметите, что каждую секунду он смещается на одно и то же расстояние в одном и том же направлении (рис. 9.2).

Теперь, когда астероиды начали двигаться, наш корабль в большой опасности — ему тоже нужно двигаться, чтобы избежать столкновения с ними. Но движение

с постоянной скоростью не спасет корабль, потому что в какой-то момент он почти наверняка столкнется с каким-нибудь астероидом. Игрок должен иметь возможность менять скорость корабля, то есть скорость и направление движения. Далее мы посмотрим, как смоделировать изменение скорости, известное как *ускорение*.

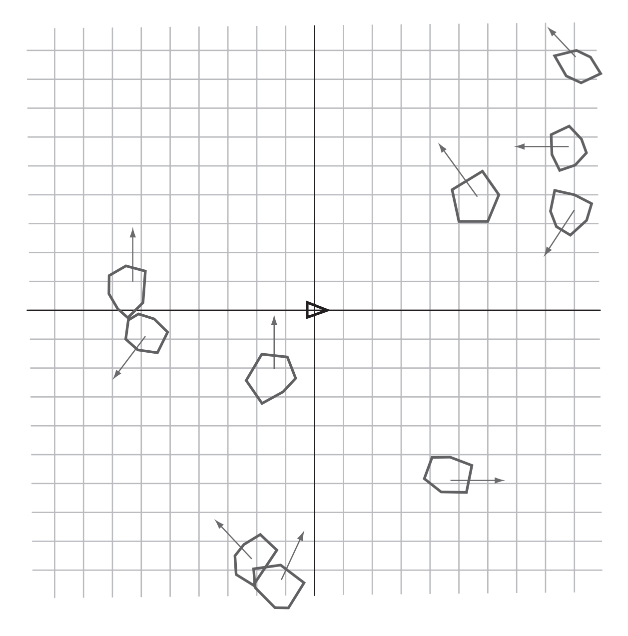


Рис. 9.2. После добавления предыдущего кода все астероиды начинают двигаться со случайной постоянной скоростью

9.1.4. Упражнения

Упражнение 9.1. Вектор скорости астероида $\mathbf{v} = (v_x, v_y) = (-3, 1)$. В каком направлении он движется на экране?

1. Вверх и вправо.
2. Вверх и влево.
3. Вниз и влево.
4. Вниз и вправо.

Решение. В данный момент $x'(t) = v_x = -3$, поэтому астероид движется в отрицательном направлении вдоль оси x, или влево; $y'(t) = v_y = 1$, соответственно, астероид движется в положительном направлении вдоль оси y, или вверх. Верный ответ 2.

9.2. МОДЕЛИРОВАНИЕ УСКОРЕНИЯ

Представим, что наш космический корабль оборудован двигателем, который, сжигая ракетное топливо, выбрасывает реактивную струю в одном направлении и толкает судно в противоположном направлении (рис. 9.3).

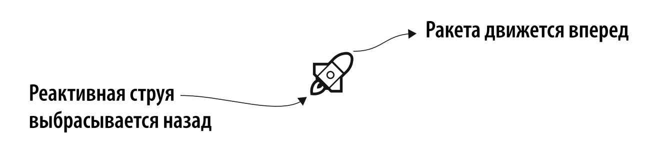


Рис. 9.3. Схема движения ракеты

Допустим, что когда работает двигатель, ракета ускоряется с постоянной скоростью в указанном направлении. Поскольку ускорение определяется как производная от скорости, постоянное значение ускорения говорит о том, что скорость вдоль обеих осей координат изменяется с постоянной скоростью. Когда ускорение отлично от нуля, скорости v_x и v_y непостоянны, они определяются функциями $v_x(t)$ и $v_y(t)$, изменяющимися во времени. Предположение о постоянстве ускорения означает, что есть два числа, a_x и a_y, такие, что $v'_x(t) = a_x$ и $v'_y(t) = a_y$. Обозначим ускорение в векторной нотации как $\mathbf{a} = (a_x, a_y)$.

Наша цель — добавить объекту — космическому кораблю — пару свойств, представляющих a_x и a_y, и заставить его ускоряться и двигаться по экрану в соответствии с этими значениями. Когда пользователь не нажимает никаких клавиш, ускорение космического корабля вдоль обеих осей координат должно быть равно нулю, а когда нажимает клавишу со стрелкой вверх, значения ускорения должны немедленно измениться и вектор (a_x, a_y) станет ненулевым, указывающим в направлении движения корабля. Пока пользователь удерживает нажатой клавишу со стрелкой вверх, скорость и положение космического корабля должны изменяться, создавая эффект перемещения.

9.2.1. Ускоренное движение космического корабля

Независимо от того, куда смотрит космический корабль, он должен ускоряться с одинаковой скоростью. То есть во время работы двигателя вектор (a_x, a_y) должен иметь фиксированное значение. Методом проб и ошибок я выяснил, что ускорение в 3 единицы делает корабль достаточно маневренным. Добавим эту константу в код игры:

```
acceleration = 3
```

Если представить, что единица расстояния в игре равна 1 м, тогда ускорение в 3 единицы будет эквивалентно 3 м/с/с (метры в секунду за секунду). Если

перед пуском двигателя космический корабль стоит на месте и игрок удерживает нажатой клавишу со стрелкой вверх, то скорость корабля будет увеличиваться на 3 м/с каждую секунду. Библиотека PyGame оперирует временем в миллисекундах, поэтому соответствующее изменение скорости будет составлять 0,003 м/с каждую миллисекунду (0,003 м/с/мс).

Теперь разберемся, как вычислить вектор ускорения $\mathbf{a} = (a_x, a_y)$ при нажатой клавише со стрелкой вверх. Если корабль направлен под углом θ, то необходимо с помощью законов тригонометрии найти вертикальную и горизонтальную составляющие ускорения по величине $|\mathbf{a}| = 3$. Согласно определению синуса и косинуса горизонтальная и вертикальная составляющие вектора скорости равны $|\mathbf{a}| \cdot \cos\theta$ и $|\mathbf{a}| \cdot \sin\theta$ соответственно (рис. 9.4). Иначе говоря, вектор ускорения определяется парой компонентов $(|\mathbf{a}| \cdot \cos\theta, |\mathbf{a}| \cdot \sin\theta)$. Кстати, получить эти компоненты из величины и направления ускорения можно с помощью функции `from_polar`, написанной в главе 2.

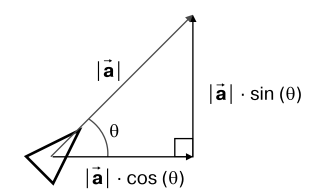


Рис. 9.4. Использование законов тригонометрии для вычисления компонентов ускорения по его величине и направлению

Во время каждой итерации игрового цикла мы можем обновлять скорость корабля перед его перемещением. За прошедшее время Δt приращение v_x будет равно $a_x \cdot \Delta t$, а приращение v_y будет равно $a_y \cdot \Delta t$. В коде нужно добавить соответствующие приращения к свойствам `vx` и `vy` корабля:

```
while not done:
    ...
        if keys[pygame.K_UP]:
            ax = acceleration * cos(ship.rotation_angle)
            ay = acceleration * sin(ship.rotation_angle)
            ship.vx += ax * milliseconds/1000
            ship.vy += ay * milliseconds/1000

        ship.move(milliseconds)
```

Определить, нажата ли клавиша со стрелкой вверх

Вычислить значения a_x и a_y на основе фиксированной величины ускорения и угла направления движения корабля

Прибавить к скоростям вдоль осей x и y приращения $a_x \cdot \Delta t$ и $a_y \cdot \Delta t$ соответственно

Переместить космический корабль, используя новые значения скоростей

Вот и все! Теперь космический корабль должен ускоряться при нажатии клавиши со стрелкой вверх. Аналогично реализуется код, управляющий поворотом при нажатии на клавиши со стрелками влево и вправо. Я не буду обсуждать здесь этот код, вы найдете его в примерах исходного кода к книге. С реализованными функциями управления вы сможете сориентировать корабль в любом направлении и придать ему дополнительное ускорение, чтобы избежать столкновения с астероидом.

Это был пример более продвинутого применения метода Эйлера при наличии *вторых производных*: $x''(t) = v'_x(t) = a_x$ и $y''(t) = v'_y(t) = a_y$. В каждой итерации

игрового цикла мы сначала обновляем скорости, затем используем результат в методе перемещения `move`, чтобы определить новые местоположения.

Мы закончили программирование игры, однако в следующих разделах продолжим обсуждение метода Эйлера и оценим, насколько хорошо он аппроксимирует движение.

9.3. БОЛЕЕ ГЛУБОКОЕ ПОГРУЖЕНИЕ В МЕТОД ЭЙЛЕРА

Основная идея метода Эйлера состоит в том, чтобы, начав с некоторого исходного значения величины (например, местоположения) и уравнения, описывающего ее производные (допустим, скорость и ускорение), изменять эту величину в соответствии с производными. Посмотрим, как это сделать, на простом примере.

Пусть некоторый объект находится в начальный момент времени $t = 0$ в местоположении (0, 0), имеет начальную скорость (1, 0) и постоянное ускорение (0, 0,2). (Для простоты в этом разделе не будем использовать единицы измерения, но вы можете продолжать думать о секундах, метрах, метрах в секунду и т. д.) Начальный вектор скорости указывает в положительном направлении вдоль оси x, а вектор ускорения — в положительном направлении вдоль оси y. То есть в первый момент объект движется точно вправо и постепенно отклоняется вверх.

Наша задача — с помощью метода Эйлера определить значения вектора местоположения через каждые 2 секунды от $t = 0$ до $t = 10$. Сначала выполним вычисления вручную, а затем реализуем их на Python. Закончив вычисления, отметим найденные позиции на плоскости xy, чтобы показать траекторию движения объекта.

9.3.1. Вычисления методом Эйлера вручную

Продолжим думать о местоположении, скорости и ускорении как о функциях времени: в любой момент объект будет иметь некоторое векторное значение для каждой из этих величин. Я дам этим векторным функциям следующие имена: $\mathbf{s}(t)$, $\mathbf{v}(t)$ и $\mathbf{a}(t)$, где $\mathbf{s}(t) = (x(t), y(t))$, $\mathbf{v}(t) = (x'(t), y'(t))$ и $\mathbf{a}(t) = (x''(t), y''(t))$. В таблице приводятся начальные значения этих функций в момент времени $t = 0$.

t	$\mathbf{s}(t)$	$\mathbf{v}(t)$	$\mathbf{a}(t)$
0	(0, 0)	(1, 0)	(0, 0,2)

В игре «Астероиды» время между циклами вычисления местоположений объектов исчислялось миллисекундами. В этом примере, чтобы побыстрее добраться

до намеченной цели, мы реконструируем местоположение объекта от момента времени $t = 0$ до $t = 10$ с шагом в 2 секунды. Далее приводится таблица, которую следует заполнить.

t	**s**(t)	**v**(t)	**a**(t)
0	(0, 0)	(1, 0)	(0, 0,2)
2			(0, 0,2)
4			(0, 0,2)
6			(0, 0,2)
8			(0, 0,2)
10			(0, 0,2)

Я уже заполнил столбец ускорения, так как мы условились, что ускорение постоянно. Что произойдет за двухсекундный интервал между $t = 0$ и $t = 2$? Скорости изменятся в соответствии с ускорением, как показано в следующей паре уравнений. В этих уравнениях снова используется греческая буква Δ для обозначения приращения переменной на рассматриваемом интервале. Например, Δt — это приращение времени, поэтому $\Delta t = 2$ с для каждого из пяти интервалов. Соответственно, компоненты скорости в момент времени 2 составят:

$$v_x(2) = v_x(0) + a_x(0) \cdot \Delta t = 1 + 0 = 1;$$

$$v_y(2) = v_y(0) + a_y(0) \cdot \Delta t = 0{,}2 \cdot 2 = 0{,}4.$$

Новое векторное значение скорости в момент времени $t = 2$ составляет $v(2) = (v_x(2), v_y(2)) = (1, 0{,}4)$. Местоположение также меняется в зависимости от скорости $v(0)$:

$$x(2) = x(0) + v_x(0) \cdot \Delta t = 0 + 1 \cdot 2 = 2;$$

$$y(2) = y(0) + v_y(0) \cdot \Delta t = 0 + 0 \cdot 2 = 0.$$

Новое значение местоположения: $\mathbf{s} = (x, y) = (2, 0)$. Это дает нам вторую строку таблицы.

t	**s**(t)	**v**(t)	**a**(t)
0	(0, 0)	(1, 0)	(0, 0,2)
2	(2, 0)	(1, 0)	(0, 0,2)
4			(0, 0,2)
6			(0, 0,2)
8			(0, 0,2)
10			(0, 0,2)

Между $t = 2$ и $t = 4$ ускорение остается неизменным, поэтому скорость увеличивается на ту же величину, $\mathbf{a} \cdot \Delta t = (0, 0{,}2) \cdot 2 = (0, 0{,}4)$, до нового значения $\mathbf{v}(4) = (1, 0{,}8)$. Местоположение меняется согласно вектору скорости $\mathbf{v}(2)$:

$$\Delta\mathbf{s} = \mathbf{v}(2) \cdot \Delta t = (1, 0{,}4) \cdot 2 = (2, 0{,}8)$$

и получает значение $\mathbf{s}(4) = (4, 0{,}8)$. Теперь у нас заполнены три строки таблицы, и мы вычислили два местоположения из пяти.

t	**s**(*t*)	**v**(*t*)	**a**(*t*)
0	(0, 0)	(1, 0)	(0, 0,2)
2	(2, 0)	(1, 0)	(0, 0,2)
4	(4, 0,8)	(1, 0,8)	(0, 0,2)
6			(0, 0,2)
8			(0, 0,2)
10			(0, 0,2)

Мы могли бы продолжать в том же духе, однако гораздо интереснее переложить всю работу на Python — это наш следующий шаг. Но прежде ненадолго приостановимся. В нескольких предыдущих абзацах я представил довольно много арифметических вычислений. Ничего из моих предположений не показалось вам подозрительным? Дам подсказку: использовать уравнение $\Delta\mathbf{s} = \mathbf{v} \cdot \Delta t$, как сделал я, здесь не совсем правильно, поэтому позиции в таблице можно назвать верными с большой натяжкой. Если вы еще не заметили, где я применил аппроксимацию, не переживайте. Как только мы нанесем векторы местоположений на график, вы это увидите сами.

9.3.2. Реализация алгоритма на Python

Описать эту процедуру на языке Python не составляет большого труда. Сначала установим начальные значения времени, местоположения, скорости и ускорения:

```
t = 0
s = (0,0)
v = (1,0)
a = (0,0.2)
```

Нас также интересуют моменты времени 0, 2, 4, 6, 8 и 10 секунд. Но вместо простого перечисления используем тот факт, что начальный момент $t = 0$, и определим константу $\Delta t = 2$ приращения времени на каждом шаге, а также количество шагов, равное 5:

```
dt = 2
steps = 5
```

Наконец, нам нужно организовать приращение времени, местоположения и скорости на каждом шаге. По мере продвижения вперед будем сохранять новые местоположения для последующего применения в массиве:

```
from vectors import add, scale
positions = [s]
for _ in range(0,5):
    t += 2
    s = add(s, scale(dt,v))

    v = add(v, scale(dt,a))
    positions.append(s)
```

Обновить местоположение, прибавив приращение Δs = v · Δt к текущим координатам s (здесь использованы функции scale и add из главы 2)

Обновить скорость, прибавив приращение Δv = a · Δt к текущей скорости v

Если запустить этот код, то он заполнит список местоположений шестью значениями вектора **s**, соответствующими моментам времени $t = 0, 2, 4, 6, 8, 10$. Теперь, получив числовые значения, можно нанести их на график и увидеть траекторию движения объекта. Нарисовав график с помощью модуля рисования из глав 2 и 3, мы увидим, что объект сначала движется вправо, а затем, как и ожидалось, поднимается вверх (рис. 9.5). Вот код на Python:

```
from draw2d import *
draw2d(Points2D(*positions))
```

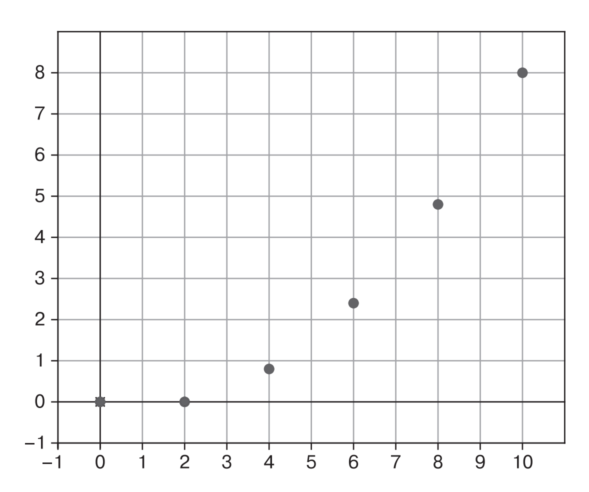


Рис. 9.5. Точки, определяющие траекторию объекта, полученные согласно вычислениям по методу Эйлера

Если верить нашей аппроксимации, объект двигался вдоль пяти прямолинейных отрезков с разной скоростью (рис. 9.6).

По условию задачи объект ускоряется постоянно, поэтому логичнее было бы увидеть траекторию в виде плавной кривой, а не ломаной линии. Теперь, когда у нас есть реализация метода Эйлера на Python, можно быстро попробовать запустить ее повторно с другими параметрами и оценить качество аппроксимации.

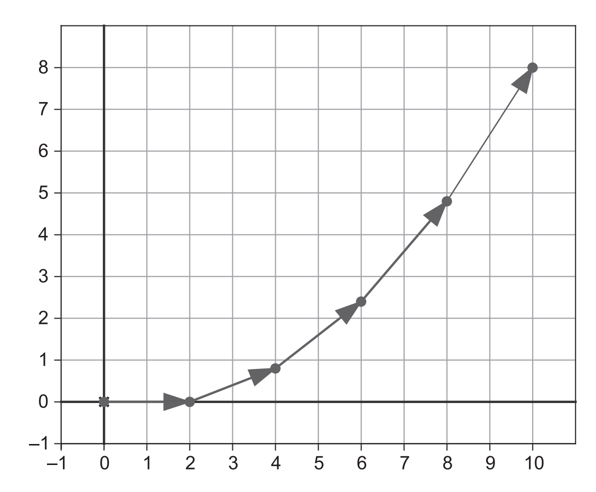


Рис. 9.6. Пять векторов, соединяющих точки на траектории

9.4. ПРИМЕНЕНИЕ МЕТОДА ЭЙЛЕРА С УМЕНЬШЕННЫМ ВРЕМЕННЫМ ШАГОМ

Повторим вычисления, использовав в два раза больше временных шагов, для чего установим константы `dt = 1` и `steps = 10`. Они по-прежнему моделируют 10-секундное перемещение объекта, но уже на 10 прямолинейных отрезках (рис. 9.7).

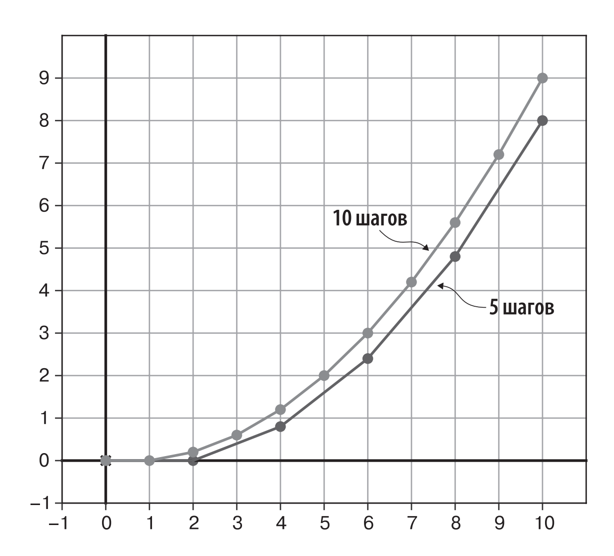


Рис. 9.7. Метод Эйлера дает разные результаты с одними и теми же начальными значениями, но с разным количеством шагов

Повторив попытку со значениями констант `dt = 0.1` и `steps = 100`, мы увидим другую траекторию перемещения объекта за те же 10 секунд (рис. 9.8).

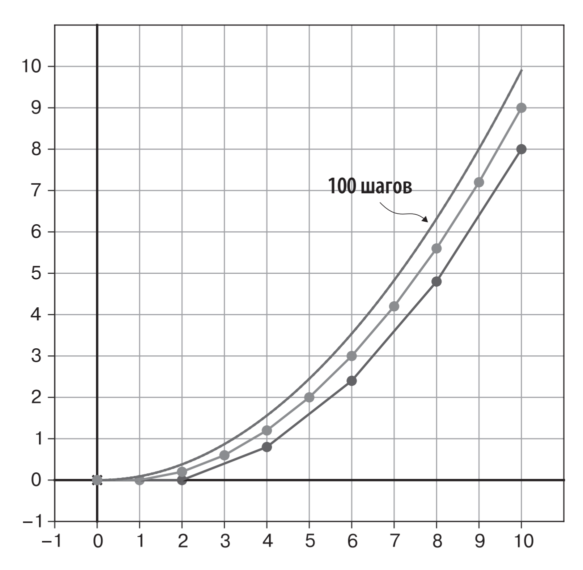


Рис. 9.8. Увеличив число шагов до 100, получим другую траекторию. Я не стал выводить точки на этой траектории, потому что их слишком много

Почему результаты получились столь разными, хотя во всех трех случаях применялись одни и те же уравнения? Судя по всему, чем больше временных шагов мы используем, тем больше становятся координаты y. Проблему легко заметить, если внимательно рассмотреть первые 2 секунды.

В аппроксимации с пятью шагами в первые 2 секунды объект продолжает двигаться вдоль оси x. В аппроксимации с 10 шагами скорость объекта обновилась раньше, поэтому он поднялся выше над осью x. Наконец, в аппроксимации со 100 шагами скорость между отметками времени $t = 0$ и $t = 1$ обновляется 19 раз, поэтому она увеличивается быстрее (рис. 9.9).

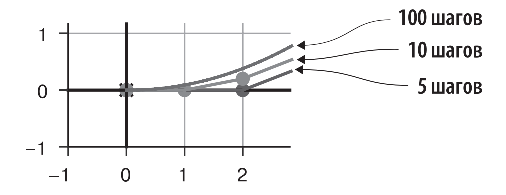


Рис. 9.9. При внимательном рассмотрении первых 2 секунд становится видно, что в аппроксимации со 100 шагами скорость увеличивается быстрее, потому что обновляется чаще

Именно это я имел в виду, когда намекал на ошибку. Уравнение $\Delta\mathbf{s} = \mathbf{v} \cdot \Delta t$ верно только тогда, когда скорость постоянна. Метод Эйлера является хорошим приближением, когда используется много временных шагов, потому

что на небольших временных интервалах скорость меняется не так сильно. Чтобы убедиться в этом, можете попробовать повторить вычисления, увеличив количество шагов и уменьшив размер одного шага *dt*. Например, если сделать 100 шагов по 0,1 секунды, конечная позиция будет иметь координаты

```
(9.99999999999998, 9.900000000000006)
```

а если сделать 100 000 шагов по 0,0001 секунды, то конечная позиция получится равной

```
(9.999999999990033, 9.999899999993497)
```

Точное значение конечной позиции равно (10,0, 10,0), и чем больше шагов будет использоваться для аппроксимации с помощью метода Эйлера, тем ближе вычисленные результаты окажутся к точным значениям. Пока вам придется поверить мне, что (10,0, 10,0) — это точное значение, но в следующей главе я расскажу, как вычислять точные интегралы, чтобы доказать это. Оставайтесь с нами!

9.4.1. Упражнения

Упражнение 9.2. Мини-проект. Напишите функцию, которая автоматически применяет метод Эйлера для вычисления местоположения объекта, движущегося с постоянным ускорением. Функция должна принимать вектор ускорения, вектор начальной скорости, вектор начального местоположения и, возможно, другие параметры.

Решение. Я также добавил в число параметров общее время и количество шагов, чтобы упростить проверку различных ответов в решении:

```
def eulers_method(s0,v0,a,total_time,step_count):
    trajectory = [s0]
    s = s0
    v = v0
    dt = total_time/step_count
    for _ in range(0,step_count):
        s = add(s,scale(dt,v))
        v = add(v,scale(dt,a))
        trajectory.append(s)
    return trajectory
```

Продолжительность каждого временного шага dt равна общему прошедшему времени, деленному на количество временных шагов

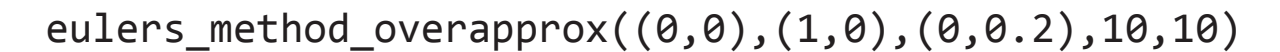
На каждом шаге вычислить новое местоположение и скорость и добавить вычисленное местоположение в конец списка, представляющего траекторию

Упражнение 9.3. Мини-проект. В разделе 9.4 вычисленная координата *y* объекта все больше отставала от истинной с течением времени, потому что компонента *y* скорости обновляется в конце каждого временного интервала. Попробуйте обновлять скорость в начале каждого временного интервала и покажите, что в этом случае координата *y*, наоборот, все больше опережает истинную.

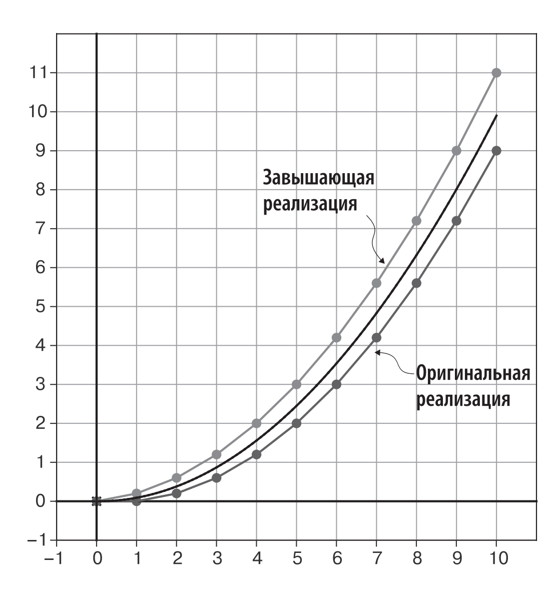
Решение. Изменим реализацию функции `eulers_method` из мини-проекта 9.2, просто переставив местами инструкции обновления `s` и `v`:

```
def eulers_method_overapprox(s0,v0,a,total_time,step_count):
    trajectory = [s0]
    s = s0
    v = v0
    dt = total_time/step_count
    for _ in range(0,step_count):
        v = add(v,scale(dt,a))
        s = add(s,scale(dt,v))
        trajectory.append(s)
    return trajectory
```

При тех же исходных данных такая аппроксимация действительно дает координату *y*, все больше опережающую истинную. Если внимательно посмотреть на траекторию, изображенную на рисунке, то можно увидеть, что объект смещается вверх уже на первом шаге.



```
eulers_method_overapprox((0,0),(1,0),(0,0.2),10,10)
```



Траектории, полученные оригинальной и новой реализациями метода Эйлера. Точная траектория показана черным цветом для сравнения

Упражнение 9.4. Мини-проект. Любой снаряд, например, брошенный бейсбольный мяч, пуля или летящий сноубордист, подвержен воздействию одного и того же вектора ускорения — 9,81 м/с/с по направлению к земле. Если представить ось x как поверхность земли и ось y, положительным концом направленную вверх, то получим вектор ускорения (0, 9,81). Если бросок бейсбольного мяча выполняется на высоте плеча в точке $x = 0$, то можно сказать, что он имеет начальное местоположение (0, 1,5). Предположим, что мяч брошен с начальной скоростью 30 м/с под углом 20° вверх в направлении положительного конца оси x. Смоделируйте его траекторию с помощью метода Эйлера. Какое примерно расстояние пролетит бейсбольный мяч вдоль оси x, прежде чем упадет на землю?

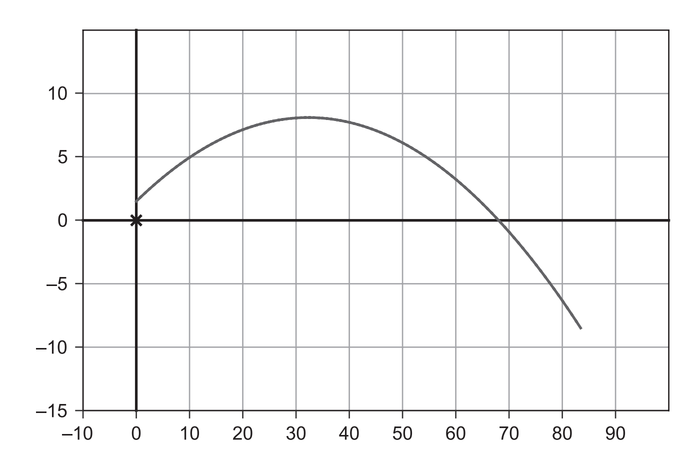
Решение. Начальная скорость равна ($30 \cdot \cos 20°$, $30 \cdot \sin 20°$). Для моделирования движения бейсбольного мяча в течение нескольких секунд можно использовать функцию `eulers_method` из упражнения 9.2:

```
from math import pi,sin,cos

angle = 20 * pi/180
s0 = (0,1.5)
v0 = (30*cos(angle),30*sin(angle))
a = (0,-9.81)

result = eulers_method(s0,v0,a,3,100)
```

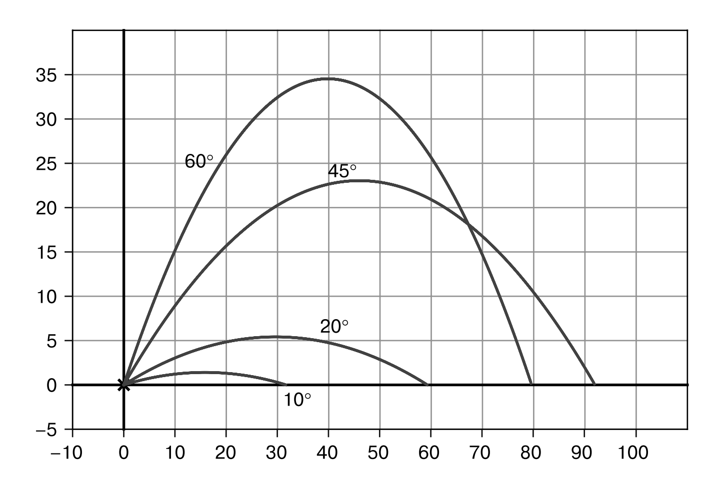
Получившаяся траектория, изображенная на рисунке, показывает, что бейсбольный мяч описывает дугу в воздухе, прежде чем упасть на землю примерно на 67-метровой отметке на оси x. Траектория продолжается ниже уровня земли, потому что мы не остановили вычисления вовремя.



Упражнение 9.5. Мини-проект. Повторно проведите моделирование методом Эйлера из предыдущего упражнения с той же начальной скоростью 30 м/с, но используя начальную позицию (0, 0) и разные углы. Под каким углом броска бейсбольный мяч пролетит дальше всего, прежде чем упадет на землю?

Решение. Для удобства моделирования можно упаковать программный код в функцию. На следующем рисунке показаны различные траектории полета мяча при броске из начальной позиции (0, 0) под разными углами. Судя по траекториям, наибольшее расстояние мяч пролетит, если бросить его под углом 45°. (Обратите внимание на то, что здесь я отфильтровал точки траектории с отрицательными компонентами y, чтобы траектории не пересекали поверхность земли.)

```
def baseball_trajectory(degrees):
    radians = degrees * pi/180
    s0 = (0,0)
    v0 = (30*cos(radians),30*sin(radians))
    a = (0,-9.81)
    return [(x,y) for (x,y) in eulers_method(s0,v0,a,10,1000) if y>=0]
```



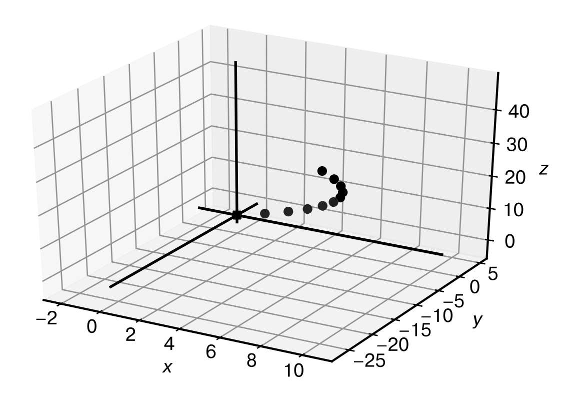
Траектории полета бейсбольного мяча, брошенного
с начальной скоростью 30 м/с под разными углами

Упражнение 9.6. Мини-проект. Объект, движущийся в трехмерном пространстве, имеет начальную скорость (1, 2, 0) и постоянный вектор ускорения (0, –1, 1). Если допустить, что движение начинается в начале координат, то где он окажется через 10 секунд? Постройте его траекторию в трехмерном пространстве, используя функции рисования из главы 3.

Решение. Оказывается, наша реализация `eulers_method` способна работать с трехмерными векторами! Траектория объекта показана на рисунке после фрагмента кода:

```
from draw3d import *

traj3d = eulers_method((0,0,0), (1,2,0), (0,-1,1), 10, 10)
draw3d(
    Points3D(*traj3d)
)
```



Разбив аппроксимацию на 1000 шагов для достижения высокой точности, получаем последнюю позицию:

```
>>> eulers_method((0,0,0), (1,2,0), (0,-1,1), 10, 1000)[-1]
(9.999999999999831, -29.949999999999644, 49.94999999999933)
```

Этот результат очень близок к точному значению конечного местоположения (10, –30, 50).

КРАТКИЕ ИТОГИ ГЛАВЫ

- Скорость — это производная местоположения по времени, вектор, состоящий из производных всех функций местоположения. В двухмерном пространстве, используя функции местоположения $x(t)$ и $y(t)$, можно записать *вектор* местоположения как функцию $\mathbf{s}(t) = (x(t), y(t))$ и вектор скорости как функцию $\mathbf{v}(t) = (x'(t), y'(t))$.
- В видеоигре можно создать анимационный эффект движения объекта с постоянной скоростью, обновляя его местоположение в каждом кадре. Произведение времени между кадрами на скорость объекта дает приращение его местоположения.

Ускорение — это производная скорости по времени, вектор, компонентами которого являются производные компонент скорости, например, $\mathbf{a}(t) = \left(v_x'(t), v_y'(t)\right)$.

- Для моделирования ускоряющегося объекта в видеоигре нужно в каждом кадре обновлять не только его местоположение, но и скорость.
- Если известна скорость, с которой величина изменяется во времени, можно вычислить значение этой величины в некоторый момент времени, рассчитав ее изменение за множество небольших временных интервалов. Это называется методом Эйлера.

10

Работа с символьными выражениями

В этой главе

- ✓ Представление алгебраических выражений как структур данных.
- ✓ Реализация программного кода для анализа, преобразования или вычисления алгебраических выражений.
- ✓ Поиск производной функции путем преобразования выражения, определяющего ее.
- ✓ Реализация функции на Python для вычисления формул производных.
- ✓ Использование библиотеки SymPy для вычисления интегралов.

Если вы опробовали все примеры кода и выполняли все упражнения в главах 8 и 9, то должны довольно хорошо разбираться в двух наиболее важных понятиях математического анализа — производной и интеграле. Во-первых, вы узнали, как аппроксимировать производную функции в точке, определяя наклоны все меньших и меньших отрезков секущих. Затем узнали, как аппроксимировать интеграл, оценивая площадь под графиком и вычисляя сумму площадей узких прямоугольников. Наконец, вы узнали, как выполнять вычисления с векторами, просто производя соответствующие операции с каждой координатой.

Кто-то может посчитать меня излишне самоуверенным, но я действительно считаю, что всего в нескольких главах этой книги мне удалось познакомить вас с самыми важными понятиями, которые изучают в течение годичного курса

математического анализа в колледже. А ларчик открывается просто: поскольку мы используем Python, я пропускаю самую трудоемкую часть традиционного курса матанализа, включающую множество манипуляций с формулами вручную. Такого рода работа позволяет взять формулу функции, такой как $f(x) = x^3$, и вычислить точную формулу ее производной $f'(x)$. В данном случае существует простой ответ: $f'(x) = 3x^2$, как показано на рис. 10.1.

Существует бесконечное количество формул, производные которых вам могут понадобиться, но вряд ли вы сможете запомнить их все, поэтому на занятиях по математическому анализу изучают небольшой набор правил и приемы их систематического применения для преобразования функции в ее производную. По большому счету, это не особенно полезный навык для программиста. Когда понадобится точная формула производной, можно воспользоваться специальным инструментом под названием *система компьютерной алгебры*, чтобы вычислить ее.

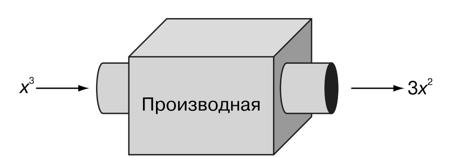


Рис. 10.1. Производная функции $f(x) = x^3$ имеет точную формулу, а именно $f'(x) = 3x^2$

10.1. ПОИСК ТОЧНОЙ ПРОИЗВОДНОЙ С ПОМОЩЬЮ СИСТЕМЫ КОМПЬЮТЕРНОЙ АЛГЕБРЫ

Одна из самых популярных систем компьютерной алгебры называется *Mathematica*. Ее движок доступен для бесплатного использования на веб-сайте Wolfram Alpha (wolframalpha.com). Как показывает мой опыт, если нужна точная формула производной для программы, которую вы пишете, лучше всего проконсультироваться с Wolfram Alpha. Например, когда мы будем строить нейронную сеть в главе 16, нам пригодится возможность узнать производную функции

$$f(x) = \frac{1}{1+e^{-x}}.$$

Чтобы найти формулу производной этой функции, зайдите на сайт wolframalpha.com и введите формулу в поле ввода (рис. 10.2). Mathematica имеет свой синтаксис записи математических формул, но сайт Wolfram Alpha удивительно дружелюбен и понимает большинство простых формул (даже записанных с использованием синтаксиса Python!).

После нажатия клавиши Enter движок Mathematica, на котором работает Wolfram Alpha, определит ряд аспектов, касающихся этой функции, включая ее производную. Если теперь прокрутить страницу вниз, то можно увидеть формулу производной функции (рис. 10.3).

WolframAlpha computational intelligence.

1/(1 + e**(-x))

Extended Keyboard Upload Examples Random

Рис. 10.2. Ввод формулы функции на сайте wolframalpha.com



Рис. 10.3. Wolfram Alpha сообщает формулу производной функции

Для функции $f(x)$ мгновенная скорость изменения при любом значении x определяется выражением

$$f'(x)=\frac{e^{-x}}{\left(1+e^{-x}\right)^2}.$$

Для понимающих идею производной и мгновенной скорости изменения научиться вводить формулы в Wolfram Alpha будет более важным навыком, чем любые другие, которые можно освоить на занятиях по математическому анализу. И все же, как бы цинично это ни звучало, о поведении конкретных функций можно многое узнать, вычислив их производные вручную. Просто нам, как профессиональным разработчикам программного обеспечения, едва ли когда-нибудь понадобится вычислять формулу производной или интеграла при наличии такого инструмента, как Wolfram Alpha.

И все же педант внутри вас может спросить: «Как Wolfram Alpha это делает?» Одно дело найти приближенную оценку производной, вычисляя наклоны секущих в различных точках графика, и совсем другое — получить точную формулу. Wolfram Alpha успешно интерпретирует введенную вами формулу, преобразует ее с помощью некоторых алгебраических манипуляций и выводит новую формулу. Такой подход, когда анализируются сами формулы, а не числа, называется *символьным программированием*.

Прагматик во мне хочет сказать вам: «Просто используйте Wolfram Alpha», — в то время как энтузиаст математики хочет научить вас брать производные и интегралы вручную, поэтому в данной главе я собираюсь пойти на компромисс. Мы займемся символьным программированием на Python, будем учиться манипулировать алгебраическими формулами напрямую и в конечном счете вычислять формулы их производных. Так я познакомлю вас с процессом нахождения формул производных, переложив большую часть работы на компьютер.

10.1.1. Выполнение символьных операций на Python

Для начала хочу продемонстрировать, как мы будем представлять формулы и манипулировать ими на Python. Возьмем для примера такую математическую функцию:

$$f(x) = (3x^2 + x) \sin (x).$$

На языке Python ее можно представить так:

```
from math import sin
def f(x):
    return (3*x**2 + x) * sin(x)
```

Этот код на Python упрощает вычисление формулы, но не позволяет выяснить что-то *о* ней. Например, мы могли бы спросить.

- Зависит ли формула от переменной x?
- Содержит ли она тригонометрическую функцию?
- Включает ли операцию деления?

Взглянув на формулу, мы можем быстро ответить на эти вопросы: да, да и нет. Но нет простого и надежного способа написать программу на Python, которая ответила бы на эти вопросы за нас. Например, крайне сложно, если вообще возможно, написать функцию `contains_division(f)`, которая принимает функцию f и возвращает значение `True`, если в ее определении имеется операция деления.

Посмотрим, где это могло бы пригодиться. Чтобы применить алгебраическое правило, нужно знать, какие операции задействуются и в каком порядке. Например, функция $f(x)$ — это произведение $\sin (x)$ на сумму, и существует хорошо известное алгебраическое правило разложения произведения на сумму, как показано на рис. 10.4.

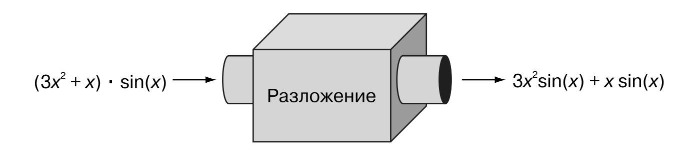


Рис. 10.4. Выражение $(3x^2 + x) \sin (x)$ является произведением числа на сумму, поэтому его можно разложить

Более верная стратегия — моделировать алгебраические выражения в виде структур данных, а не преобразовывать их непосредственно в код на Python, тогда ими будет проще манипулировать. Научившись манипулировать символьными представлениями функций, мы сможем автоматизировать применение правил математического анализа.

Производные большинства функций, выраженных простыми формулами, тоже можно выразить простыми формулами. Например, производная x^3 равна $3x^2$, то есть при любом значении x производная $f(x) = x^3$ равна $3x^2$. К концу этой главы вы научитесь писать функции на Python, которые принимают алгебраические выражения и возвращают выражения их производных. Наша структура данных для представления алгебраической формулы сможет представлять переменные, числа, суммы, разности, произведения, частные, степени и специальные функции, такие как синус и косинус. С помощью этого набора строительных блоков мы сможем представить огромное разнообразие формул, и наш механизм определения производных будет правильно обрабатывать их все (рис. 10.5).

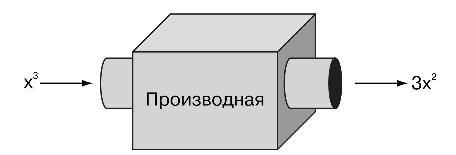


Рис. 10.5. Цель состоит в том, чтобы написать на Python функцию определения производной, которая принимает выражение с математической функцией и возвращает выражение с производной

Сначала мы рассмотрим моделирование выражений в виде структур данных на Python. Затем для разогрева выполним несколько простых вычислений со структурами данных и реализуем такие действия, как подстановка чисел вместо переменных или разложение произведения на сумму. После этого я представлю некоторые правила получения производных по формулам, затем мы напишем свою функцию определения производной и попробуем автоматически применять ее к символьным структурам данных.

10.2. МОДЕЛИРОВАНИЕ АЛГЕБРАИЧЕСКИХ ВЫРАЖЕНИЙ

Сосредоточимся на функции $f(x) = (3x^2 + x) \sin(x)$ и посмотрим, как разбить ее на части. Это хороший пример функции, потому что она содержит множество разных строительных блоков: переменную x, числа, операции сложения, умножения, возведения в степень и функцию со специальным именем $\sin(x)$. Выработав стратегию разбиения этой функции на концептуальные части, мы сможем преобразовать ее в структуру данных на Python. Эта структура данных будет играть роль *символьного* представления функции, отличного от строкового, такого как `"(3 * x ** 2 + x) * sin(x)"`.

Первое наблюдение состоит в том, что f — это произвольное имя функции. Например, разложение правой части этого уравнения выполняется одинаково

независимо от имени функции. Соответственно, мы можем сосредоточиться только на выражении, определяющем функцию, в данном случае $(3x^2 + x) \sin(x)$. Эта часть называется выражением, в отличие от уравнения, которое должно содержать знак равенства (=). *Выражение* — это набор математических символов (чисел, букв, операций и т. д.), объединенных определенным образом. Поэтому наша первая цель — смоделировать эти символы и допустимые средства составления выражения на Python.

10.2.1. Разбиение выражения на части

Начнем моделирование алгебраических выражений с их разбиения на части. Выражение $(3x^2 + x) \sin(x)$ можно разбить только одним допустимым способом — на сомножители $(3x^2 + x)$ и $\sin(x)$, как показано на рис. 10.6.



Рис. 10.6. Допустимое разбиение алгебраического выражения на два меньших выражения

Это выражение нельзя разбить на слагаемые по знаку «плюс». Конечно, выражения, получившиеся в результате разбиения, останутся допустимыми, но результат такой суммы не будет совпадать со значением исходного выражения (рис. 10.7).

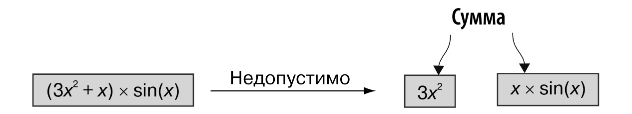


Рис. 10.7. Недопустимое разбиение алгебраического выражения по знаку «плюс», потому что исходное выражение не является суммой слагаемых $3x^2$ и $x \cdot \sin(x)$

Выражение $3x^2 + x$ можно разбить на слагаемые: $3x^2$ и x. Точно так же правила первоочередности выполнения операций говорят, что $3x^2$ — это произведение 3 и x^2, а не произведение $3x$, возведенное в степень 2.

В этой главе мы будем рассуждать о таких операциях, как умножение и сложение, как о способе соединения двух или более алгебраических выражений с целью получить новое, более крупное алгебраическое выражение. Точно так же операторы являются границами разбиения существующего алгебраического выражения на более мелкие.

В терминологии функционального программирования функции, объединяющие меньшие объекты в крупные, часто называются *комбинаторами*. Вот некоторые из комбинаторов в нашем выражении:

- $3x^2$ — *произведение* выражений 3 и x^2;
- x^2 — *возведение в степень*: выражение x возводится в степень другого выражения 2;
- выражение sin (x) — *применение функции*; объединяя выражение sin и выражение x, можно построить новое выражение sin (x).

Переменная x, число 2 или функция sin не могут быть разбиты на более мелкие части. Чтобы отличить от комбинаторов, будем называть их *элементами*. Суть заключается в том, что хотя ($3x^2 + x$) sin (x) — это просто набор символов, напечатанных на книжной странице, они комбинируются определенным образом для передачи некоторого математического смысла. Чтобы воплотить эту идею в жизнь, мы можем представить, как это выражение конструируется из основных элементов.

10.2.2. Конструирование дерева выражения

Элементов 3, x, 2 и sin вместе с комбинаторами сложения, умножения, возведения в степень и применения функции достаточно, чтобы сконструировать выражение ($3x^2 + x$) sin (x). Пройдемся по шагам и нарисуем структуру, которую мы в итоге построим. Одна из первых конструкций, которую можно составить, — это x^2, которая объединяет x и 2 с использованием комбинатора возведения в степень (рис. 10.8).

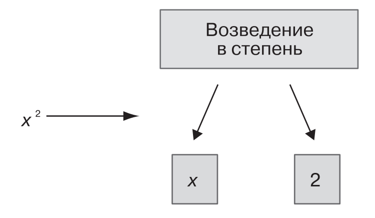


Рис. 10.8. Объединение x и 2 с помощью комбинатора возведения в степень для представления большего выражения x^2

Следующий шаг — объединение x^2 с числом 3 с использованием комбинатора умножения, дающее в результате выражение $3x^2$ (рис. 10.9).

Эта конструкция имеет два слоя: одно из выражений, объединяемых комбинатором умножения, само является комбинатором. По мере добавления в выражение новых членов оно становится еще глубже. Следующий шаг — объединение элемента x и выражения $3x^2$ с использованием комбинатора сложения (рис. 10.10).

Наконец, задействуем комбинатор применения функции, чтобы применить sin к x, а затем комбинатор умножения, чтобы объединить sin (x) с тем, что мы построили до сих пор (рис. 10.11).

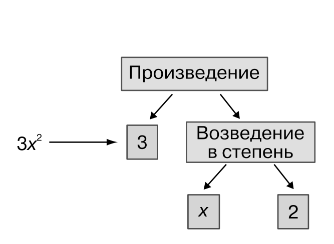


Рис. 10.9. Объединение числа 3 со степенью для получения произведения $3x^2$

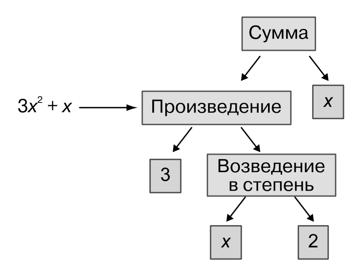


Рис. 10.10. Объединение выражения $3x^2$ и элемента x с помощью комбинатора сложения дает выражение $3x^2 + x$

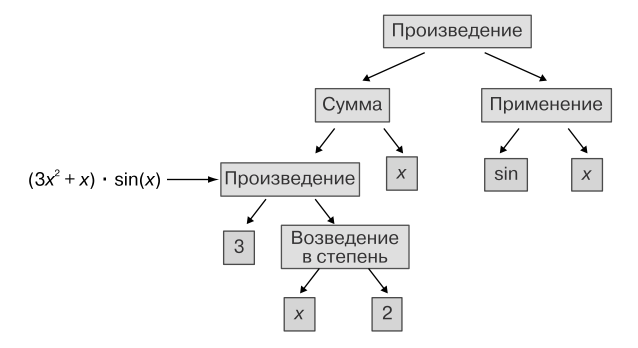


Рис. 10.11. Окончательная схема, показывающая, как скомпоновать $(3x^2 + x)$ sin (x) из элементов и комбинаторов

Построенная нами структура называется *деревом*. Корнем дерева является комбинатор умножения. Из него выходят две ветви: `Sum` (сумма) и `Apply` (применение функции). Каждый комбинатор, расположенный ниже по дереву, добавляет дополнительные ветви. Так продолжается, пока на концах всех ветвей не окажутся элементы, которые являются *листьями*. Любое алгебраическое выражение, сконструированное с использованием чисел, переменных и именованных функций в качестве элементов и операций в качестве комбинаторов, соответствует определенному дереву, раскрывающему его структуру. Следующее, что можно сделать, — построить такое же дерево на Python.

10.2.3. Представление дерева выражений на Python

Построив это дерево на Python, мы достигнем цели — представим выражение в виде структуры данных. Для представления каждого типа элементов и каждого комбинатора я буду использовать классы, описанные в приложении Б. По мере продвижения мы будем совершенствовать эти классы, наделяя их все большей

и большей функциональностью. Вы можете следовать за описанием постепенно, применяя блокнот Jupyter для главы 10, или взять законченную реализацию в файле сценария на Python `expressions.py`.

В своей реализации мы смоделируем комбинаторы как контейнеры, содержащие соответствующие входные данные. Например, возведение x в степень 2, или x^2, имеет два входных компонента: основание x и степень 2. Вот класс на Python, предназначенный для представления выражения возведения в степень:

```
class Power():
    def __init__(self,base,exponent):
        self.base = base
        self.exponent = exponent
```

Теперь можем представить выражение x^2, записав его как `Power("x",2)`. Но взамен простых строк и чисел я создам специальные классы, представляющие числа и переменные, например:

```
class Number():
    def __init__(self,number):
        self.number = number

class Variable():
    def __init__(self,symbol):
        self.symbol = symbol
```

Это может показаться ненужным излишеством, но иногда полезно уметь отличать элемент `Variable("x")`, означающий букву x, представляющую переменную, от строки `"x"`, которая является просто строкой. Используя эти три класса, можно смоделировать выражение x^2, как

```
Power(Variable("x"),Number(2))
```

Каждый из комбинаторов можно реализовать как класс с соответствующим именем, хранящий данные любых объединяемых им выражений. Например, комбинатор умножения можно представить как класс, хранящий два выражения, предназначенные для перемножения:

```
class Product():
    def __init__(self, exp1, exp2):
        self.exp1 = exp1
        self.exp2 = exp2
```

Произведение $3x^2$ можно выразить в виде следующего комбинатора:

```
Product(Number(3),Power(Variable("x"),Number(2)))
```

После добавления прочих необходимых классов мы сможем смоделировать исходное выражение, а также множество других выражений (Обратите внимание

на то, что на вход комбинатора Sum можно передать любое количество выражений. Мы могли бы точно так же поступить с комбинатором Product. Но я преднамеренно ограничил количество входных выражений в комбинаторе Product, чтобы упростить код, когда мы начнем вычислять производные в разделе 10.3.):

```
class Sum():
    def __init__(self, *exps):
        self.exps = exps

class Function():
    def __init__(self,name):
        self.name = name

class Apply():
    def __init__(self,function,argument):
        self.function = function
        self.argument = argument

f_expression = Product(
               Sum(
                   Product(
                       Number(3),
                       Power(
                           Variable("x"),
                           Number(2))),
                   Variable("x")),
               Apply(
                   Function("sin"),
                   Variable("x")))
```

Позволяет суммировать любое количество слагаемых, то есть он может сложить два или более выражений вместе

Хранит строку с именем функции (например, sin)

Хранит функцию и аргумент, к которому она применяется

Здесь я использовал дополнительные отступы, чтобы сделать структуру выражения более ясной

Это точное представление исходного выражения $(3x^2 + x) \sin(x)$. Под этим я подразумеваю, что мы можем взглянуть на этот объект Python и увидеть, что он описывает именно это алгебраическое выражение, а не какое-то другое. Взяв другое выражение, например

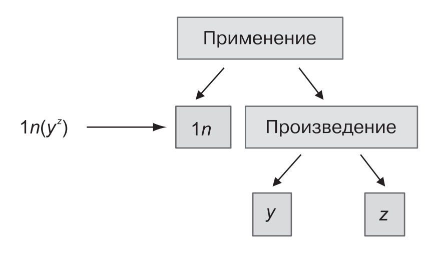
```
Apply(Function("cos"),Sum(Power(Variable("x"),Number("3")), Number(-5)))
```

мы можем внимательно прочитать его и увидеть, что оно представляет другую формулу — $\cos(x^3 + (-5))$. В следующих упражнениях вы сможете попрактиковаться в преобразовании некоторых алгебраических выражений в дерево объектов на языке Python и наоборот. Вы наверняка заметите, что ввод с клавиатуры полного представления выражения — довольно утомительное занятие. Но есть и хорошая новость: после ввода представления выражения на языке Python ручная работа закончится. В следующем разделе я покажу, как писать функции на Python, автоматизирующие работу с выражениями.

10.2.4. Упражнения

Упражнение 10.1. Возможно, вы встречались с *натуральным логарифмом* — специальной математической функцией, которая записывается ln (x). Представьте выражение ln(y^z) в виде дерева, построенного из элементов и комбинаторов, описанных в предыдущем разделе.

Решение. Самый внешний комбинатор — это `Apply`, применение функции. Применяемая функция — натуральный логарифм ln, а аргумент — y^z. В свою очередь y^z — это степень с основанием y и показателем z. Вот как выглядит результат.



Упражнение 10.2. Преобразуйте выражение из предыдущего упражнения в код на Python, учитывая, что натуральный логарифм вычисляется функцией `math.log`. Запишите его в двух представлениях: в виде функции на Python и в виде структуры данных, построенной из элементов и комбинаторов.

Решение. Выражение ln (y^z) можно рассматривать как функцию двух переменных, y и z. Вот как можно представить его на языке Python, где `log` представляет функцию ln:

```
from math import log
def f(y,z):
    return log(y**z)
```

В виде дерева это же выражение выглядит так:

```
Apply(Function("ln"), Power(Variable("y"), Variable("z")))
```

Упражнение 10.3. Какое выражение представлено структурой данных `Product(Number(3),Sum(Variable("y"),Variable("z")))`?

Решение. Эта структура данных представляет выражение $3 \cdot (y + z)$. Обратите внимание на то, что скобки необходимы для точного определения порядка выполнения операций.

Упражнение 10.4. Реализуйте комбинатор `Quotient`, представляющий деление одного выражения на другое. Как бы вы представили выражение

$$\frac{a+b}{2}?$$

Решение. Комбинатор `Quotient` должен хранить два выражения: верхнее — *числитель* (`numerator`) и нижнее — *знаменатель* (`denominator`):

```
class Quotient():
    def __init__(self,numerator,denominator):
        self.numerator = numerator
        self.denominator = denominator
```

Выражение в задании — это результат деления суммы $a + b$ на число 2:

```
Quotient(Sum(Variable("a"),Variable("b")),Number(2))
```

Упражнение 10.5. Реализуйте комбинатор `Difference`, представляющий разность двух выражений. Как бы вы представили выражение $b^2 - 4ac$?

Решение. Комбинатор `Difference` должен хранить два выражения: первое — уменьшаемое и второе — вычитаемое:

```
class Difference():
    def __init__(self,exp1,exp2):
        self.exp1 = exp1
        self.exp2 = exp2
```

Выражение $b^2 - 4ac$ — это разность выражений b^2 и $4ac$, которую можно представить так:

```
Difference(
    Power(Variable('b'),Number(2)),
    Product(Number(4),Product(Variable('a'), Variable('c'))))
```

Упражнение 10.6. Реализуйте комбинатор `Negative`, представляющий отрицание. Например, отрицание выражения $x^2 + y$ равно $-(x^2 + y)$. Представьте последнее выражение в виде программного кода, используя новый комбинатор.

Решение. Комбинатор `Negative` — это класс, содержащий одно выражение:

```
class Negative():
    def __init__(self,exp):
        self.exp = exp
```

Чтобы получить отрицание выражения $x^2 + y$, его нужно передать конструктору `Negative`:

```
Negative(Sum(Power(Variable("x"),Number(2)),Variable("y")))
```

Упражнение 10.7. Добавьте функцию `Sqrt`, представляющую извлечение квадратного корня, и используйте ее для создания представления формулы

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Решение. Чтобы не вводить слишком много кода, можно заранее определить переменные и функцию извлечения квадратного корня:

```
A = Variable('a')
B = Variable('b')
C = Variable('c')
Sqrt = Function('sqrt')
```

Теперь остается только преобразовать алгебраическое выражение в соответствующую структуру элементов и комбинаторов. На самом верхнем уровне находится результат деления суммы (в числителе) на произведение (в знаменателе):

```
Quotient(
    Sum(
        Negative(B),
        Apply(
            Sqrt,
            Difference(
                Power(B,Number(2)),
                Product(Number(4), Product(A,C))))),
    Product(Number(2), A))
```

Упражнение 10.8. Мини-проект. Создайте абстрактный базовый класс `Expression` и используйте его в качестве родительского во всех классах элементов и комбинаторов. Объявление класса `Variable()` в этом случае будет выглядеть как `Variable(Expression)`. Затем добавьте в него перегруженные реализации арифметических операций Python `+`, `-`, `*` и `/`, чтобы они создавали объекты `Expression`. Например, выражение `2*Variable("x")+3` должно давать `Sum(Product(Number(2),Variable("x")),Number(3))`.

Решение. Пример реализации вы найдете в файле `expressions.py` в примерах исходного кода для этой главы.

10.3. ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ СИМВОЛЬНЫХ ВЫРАЖЕНИЙ

Для функции $f(x) = (3x^2 + x) \sin (x)$ мы написали функцию на Python, которая ее вычисляет:

```
def f(x):
    return (3*x**2 + x)*sin(x)
```

Как код на языке Python эта функция хороша только для одного — вычисления значения функции для заданного входного значения x. Значение f в Python не позволяет программно ответить на вопросы, которые были заданы в начале главы: зависит ли f от входных параметров, содержит ли f тригонометрическую функцию и как будет выглядеть тело f, если его разложить алгебраически. В этом разделе мы увидим, что после преобразования выражения в структуру данных на Python, состоящую из элементов и комбинаторов, появляется возможность ответить на все эти и другие вопросы!

10.3.1. Поиск всех переменных в выражении

Напишем функцию, которая принимает выражение и возвращает список различных переменных, присутствующих в нем. Например, в определении $h(z) = 2z + 3$ используется одна входная переменная z, а в $g(x) = 7$ нет ни одной переменной. Наша функция, назовем ее `distinct_variables`, будет принимать выражение, то есть любой из элементов или комбинаторов, и возвращать множество Python, содержащее переменные.

Если выражение представлено элементом, таким как z или 7, ответ очевиден. Выражение, представленное переменной, содержит одну переменную, а выражение,

представленное числом, вообще не содержит переменных. От функции ожидается следующее поведение:

```
>>> distinct_variables(Variable("z"))
{'z'}
>>> distinct_variables(Number(3))
set()
```

Ситуация усложняется, когда выражение включает комбинаторы, например, $y \cdot z + x^z$. Человек легко прочитает все переменные y, z и x, но как их извлечь из представления выражения на Python? На самом деле это комбинатор `Sum`, представляющий сумму $y \cdot z$ и x^z. Первое выражение содержит переменные y и z, а второе — x и z. То есть сумма содержит все переменные, имеющиеся в этих двух выражениях.

Это предполагает рекурсивное решение: получив комбинатор, `distinct_variables` должна применить `distinct_variables` к каждому из содержащихся в нем выражений. В итоге функция доберется до элементов (переменных и чисел), каждый из которых содержит ноль или одну переменную. Соответственно, функция `distinct_variables` должна отдельно обрабатывать элементы и комбинаторы разных типов, составляющие выражение:

```
def distinct_variables(exp):
    if isinstance(exp, Variable):
        return set(exp.symbol)
    elif isinstance(exp, Number):
        return set()
    elif isinstance(exp, Sum):
        return set().union(*[distinct_variables(exp) for exp in exp.exps])
    elif isinstance(exp, Product):
        return distinct_variables(exp.exp1).union(distinct_variables(exp.exp2))
    elif isinstance(exp, Power):
        return distinct_variables(exp.base).union(distinct_variables(exp.exponent))
    elif isinstance(exp, Apply):
        return distinct_variables(exp.argument)
    else:
        raise TypeError("Not a valid expression.")
```

Код получился громоздким, но это всего лишь длинный оператор `if/else` с одной ветвью для каждого возможного элемента или комбинатора. Возможно, с точки зрения стиля программирования, было бы лучше добавить метод `distinct_variables` в каждый класс элемента и комбинатора, но тогда общую логику будет сложно проследить в одном листинге. Как и ожидалось, выражение `f_expression` содержит только переменную x:

```
>>> distinct_variables(f_expression)
{'x'}
```

Если вы знакомы с древовидными структурами данных, то без труда узнаете в этой функции рекурсивный обход дерева. К моменту завершения функция

вызовет `distinct_variables` для всех подвыражений, содержащихся во входном выражении, которые являются узлами дерева. Это гарантирует, что не будет пропущена ни одна переменная и на выходе получится правильный результат. В упражнениях в конце этого раздела вы сможете применить аналогичный подход и найти все числа или все функции.

10.3.2. Вычисление выражения

Теперь у нас есть два представления одной и той же математической функции $f(x)$. Одно из них — функция `f` на Python, которую удобно использовать для вычисления функции с заданном входным значением x. Второе — древовидная структура данных, которая описывает конструкцию выражения, определяющего $f(x)$. Причем последнее представление сочетает в себе лучшее из обоих миров: его также можно задействовать для вычисления $f(x)$, правда для этого нужно приложить небольшие усилия.

Механически вычисление функции $f(x)$, скажем, при $x = 5$ означает подстановку значения 5 вместо x и выполнение арифметических действий. Для простого выражения $f(x) = x$, подстановка $x = 5$ дала бы $f(5) = 5$. Другой простой пример — $g(x) = 7$, где подстановка 5 вместо x не дает никакого эффекта: в правой части переменная x отсутствует, поэтому результат $g(5)$ — это число 7, что неверно.

Код вычисления выражений в символьном представлении похож на код поиска переменных, который мы только что написали. Только вместо поиска переменных в каждом подвыражении нужно вычислить каждое подвыражение, а комбинаторы сообщат нам, как объединить результаты, чтобы получить значение всего выражения.

Для начала потребуются данные, описывающие заменяемые переменные и заменяющие их значения. Чтобы вычислить результат выражения с двумя переменными, такого как $z(x, y) = 2xy^3$, нужны два значения, например, $x = 3$ и $y = 2$. В терминологии информатики эти равенства называются *привязками переменных*. С их помощью можно вычислить подвыражение y^3 как $(2)^3$, что равно 8. Другое подвыражение, $2x$, дает в результате $2 \cdot (3) = 6$. Эти два выражения объединяются комбинатором `Product`, соответственно, значение всего выражения будет произведением 6 на 8, или 48.

Когда мы будем воплощать эту процедуру в код на Python, я покажу немного иной стиль, чем в предыдущем примере. Вместо создания отдельной функции оценки выражения мы добавим метод `evaluate` в каждый класс, представляющий выражение. Для этого определим абстрактный базовый класс `Expression` с абстрактным методом `evaluate` и унаследуем его во всех классах выражений. Если вам нужно освежить в памяти, что такое абстрактные базовые классы в Python, то отвлекитесь ненадолго и посмотрите, как мы определили класс

Vector в главе 6, или загляните в приложение Б. Вот базовый класс Expression с методом evaluate:

```
from abc import ABC, abstractmethod

class Expression(ABC):
    @abstractmethod
    def evaluate(self, **bindings):
        pass
```

Поскольку выражение может содержать несколько переменных, я объявил метод так, чтобы ему можно было передавать привязки переменных в форме именованных аргументов. Например, привязки {"x": 3, "y": 2} означают замену x на 3 и y на 2. Это дает нам хороший синтаксический сахар при вычислении выражения. Если допустить, что z представляет выражение $2xy^3$, то, закончив реализацию, мы сможем выполнить такой вызов:

```
>>> z.evaluate(x=3,y=2)
48
```

Итак, мы определили абстрактный класс. Теперь нужно унаследовать его во всех наших классах выражений. Так, экземпляр Number — это допустимое выражение, когда оно — обычное число, например 7. Независимо от заданных на входе привязок переменных число вычисляется как само это число:

```
class Number(Expression):
    def __init__(self,number):
        self.number = number
    def evaluate(self, **bindings):
        return self.number
```

Например, вычисление Number(7).evaluate(x=3,y=6,q=-15) даст в результате само число 7.

Переменные обрабатываются так же просто. Встретив выражение Variable("x"), мы должны найти соответствующую привязку и вернуть число, заданное для переменной x. Закончив реализацию, сможем выполнить вызов Variable("x").evaluate(x=5) и получить в результате 5. Если привязка для x не будет найдена, то мы не сможем завершить оценку, и тогда следует вызвать исключение. Вот обновленное определение класса Variable:

```
class Variable(Expression):
    def __init__(self,symbol):
        self.symbol = symbol
    def evaluate(self, **bindings):
        try:
            return bindings[self.symbol]
        except:
            raise KeyError("Variable '{}' is not bound.".format(self.symbol))
```

Разобравшись с этими элементами, перейдем к комбинаторам. (Обратите внимание на то, что мы не будем рассматривать объект `Function` как самостоятельное выражение, потому что такая функция, как `sine`, не является самостоятельным выражением. Ее можно вычислить, только передав аргумент в контексте комбинатора `Apply`.) Такой комбинатор, как `Product`, вычисляется просто: нужно вычислить оба выражения, составляющие произведение, а затем перемножить результаты. В произведении не требуется выполнять подстановку, но мы должны передать привязки обоим подвыражениям на тот случай, если они содержат переменные:

```
class Product(Expression):
    def __init__(self, exp1, exp2):
        self.exp1 = exp1
        self.exp2 = exp2
    def evaluate(self, **bindings):
        return self.exp1.evaluate(**bindings) * self.exp2.evaluate(**bindings)
```

С этими тремя классами, дополненными методами вычисления, можем вычислить любое выражение, сконструированное из переменных, чисел и произведений, например,

```
>>> Product(Variable("x"), Variable("y")).evaluate(x=2,y=5)
10
```

Точно так же можно добавить метод `evaluate` в комбинаторы `Sum`, `Power`, `Difference` и `Quotient` и в любые другие комбинаторы, которые вы, возможно, создали упражняясь. Вычислив подвыражения, к полученным результатам можно применить соответствующую операцию и найти общий результат.

Комбинатор `Apply` работает немного иначе, поэтому уделим ему особое внимание. Нам нужно отыскать функцию с заданным именем, например `sin` или `sqrt`, и выяснить, как вычислить ее значение. Есть несколько способов сделать это, но я предпочитаю сохранять известные функции в словаре и использовать его в классе `Apply`. На первом этапе можем поместить в словарь три функции:

```
_function_bindings = {
    "sin": math.sin,
    "cos": math.cos,
    "ln": math.log
}

class Apply(Expression):
    def __init__(self,function,argument):
        self.function = function
        self.argument = argument
    def evaluate(self, **bindings):
        return
_function_bindings[self.function.name](self.argument.evaluate(**bindings))
```

Вы можете попробовать самостоятельно реализовать методы `evaluate` в остальных классах или найти их в примерах исходного кода для этой книги. Реализовав их, вы сможете оценить выражение `f_expression` из раздела 10.1.3:

```
>>> f_expression.evaluate(x=5)
-76.71394197305108
```

Результат здесь не важен, важно лишь, что он совпадает с результатом, возвращаемым обычной функцией $f(x)$ на Python:

```
>>> f(5)
-76.71394197305108
```

Снабженные методом `evaluate`, наши объекты `Expression` могут выполнять ту же работу, что и соответствующие им обычные функции на Python.

10.3.3. Разложение выражения

Со структурами данных, представляющими выражения, можно многое сделать. В упражнениях вам будет предложено попробовать свои силы при создании еще нескольких функций на Python, манипулирующих выражениями. А пока я покажу еще один пример, о котором упоминал в начале этой главы, — разложение выражения. Под этим я подразумеваю выполнение любого произведения или возведение сумм в степень.

Будем руководствоваться алгебраическим правилом, основанным на *распределительном свойстве* сумм и произведений. Согласно этому правилу произведение вида $(a + b) \cdot c$ равно сумме произведений $ac + bc$, аналогично, $x(y + z) = xy + xz$. Например, выражение $(3x^2 + x) \sin(x)$ равно выражению $3x^2 \sin(x) + x \sin(x)$, которое называется развернутой формой первого произведения. Это правило можно применить столько раз, сколько потребуется, чтобы разложить более сложные выражения, например:

$$
\begin{gathered}
(x + y)^3 = (x + y)(x + y)(x + y) = \\
= x(x + y)(x + y) + y(x + y)(x + y) = \\
= x^2(x + y) + xy(x + y) + yx(x + y) + y^2(x + y) = \\
= x^3 + x^2y + x^2y + xy^2 + yx^2 + y^2x + y^2x + y^3 = \\
= x^3 + 3x^2y + 3y^2x + y^3.
\end{gathered}
$$

Как видите, разложение короткого выражения, такого как $(x + y)^3$, может потребовать много времени. В дополнение к разложению этого выражения я также немного упростил результат, например, представив некоторые произведения, которые выглядели как xyx или xxy, в форме x^2y. Это вполне допустимо,

потому что от перестановки мест сомножителей произведение не меняется. Затем упростил выражение еще больше, объединив одинаковые члены, такие как x^2y и y^2x, встречающиеся по три раза, и представив их все вместе в виде $3x^2y$ и $3y^2x$. В следующем примере мы посмотрим только, как реализовать разложение, а выполнить упрощение можете сами в качестве самостоятельного упражнения.

Начнем с добавления абстрактного метода `expand` в базовый класс `Expression`:

```
class Expression(ABC):
    ...
    @abstractmethod
    def expand(self):
        pass
```

Если выражение — это переменная или число, то оно уже разложено. В этих случаях метод `expand` должен возвращать сам объект, например:

```
class Number(Expression):
    ...
    def expand(self):
        return self
```

Суммы считаются уже разложенными выражениями, но отдельные слагаемые суммы могут быть не разложены. Например, $5 + a(x + y)$ — это сумма, в которой первый член 5 полностью разложен, а второй член $a(x + y)$ — нет. Чтобы разложить сумму, нужно разложить каждое из слагаемых и объединить их в сумму:

```
class Sum(Expression):
    ...
    def expand(self):
        return Sum(*[exp.expand() for exp in self.exps])
```

То же относится к применению функции. Мы не можем разложить саму функцию, но можем разложить ее аргументы. Например, превратит выражение $\sin(x(y + z))$ в $\sin(xy + xz)$ такое разложение:

```
class Apply(Expression):
    ...
    def expand(self):
        return Apply(self.function, self.argument.expand())
```

Настоящая работа начинается, когда дело доходит до разложения произведения или возведения в степень, потому что в этих случаях структура выражения меняется полностью. Например, выражение $a(b + c)$ — это произведение переменной на сумму двух переменных, а его развернутая форма $ab + ac$ — сумма двух произведений двух переменных. Чтобы реализовать распределительный закон, нужно предусмотреть три случая: первый член произведения — это сумма,

второй член — это сумма или ни один из членов не является суммой. В последнем случае разложение не требуется:

```
class Product(Expression):
    ...
    def expand(self):
        expanded1 = self.exp1.expand()
        expanded2 = self.exp2.expand()
        if isinstance(expanded1, Sum):
            return Sum(*[Product(e,expanded2).expand()
                         for e in expanded1.exps])
        elif isinstance(expanded2, Sum):
            return Sum(*[Product(expanded1,e)
                         for e in expanded2.exps])
        else:
            return Product(expanded1,expanded2)
```

Разложить оба множителя

Если первый множитель — сумма, то каждое из его слагаемых нужно умножить на второй множитель, а затем разложить результат, если второй множитель тоже сумма

Если второй множитель — сумма, то каждое из его слагаемых нужно умножить на первый множитель

Иначе ни один из множителей не является суммой и не требует разложения

После реализации всех этих методов можно протестировать функцию `expand`. Соответствующим образом реализовав `__repr__` (см. упражнения), можно получить ясное строковое представление результатов в Jupyter или в интерактивном сеансе Python. Функция правильно преобразует $(a + b)(x + y)$ в $ax + ay + bx + by$:

```
Y = Variable('y')
Z = Variable('z')
A = Variable('a')
B = Variable('b')

>>> Product(Sum(A,B),Sum(Y,Z))
Product(Sum(Variable("a"),Variable("b")),Sum(Variable("x"),Variable("y")))
>>> Product(Sum(A,B),Sum(Y,Z)).expand()
Sum(Sum(Product(Variable("a"),Variable("y")),Product(Variable("a"),
Variable("z"))),Sum(Product(Variable("b"),Variable("y")),
Product(Variable("b"),Variable("z"))))
```

И выражение $(3x^2 + x) \sin(x)$ тоже правильно преобразуется в $3x^2 \sin(x) + x \sin(x)$:

```
>>> f_expression.expand()
Sum(Product(Product(3,Power(Variable("x"),2)),Apply(Function("sin"),Variable(
"x"))),Product(Variable("x"),Apply(Function("sin"),Variable("x"))))
```

К данному моменту мы написали на Python несколько функций, которые преобразуют выражения, используя законы алгебры, а не только арифметики. Этот вид программирования, называемый *символьным программированием*, или, точнее, *компьютерной алгеброй*, имеет множество захватывающих применений, но я не смогу охватить их все в рамках этой книги. Обязательно попробуйте свои силы в решении нескольких следующих упражнений, а затем мы перейдем к самому важному примеру — поиску формул производных.

10.3.4. Упражнения

Упражнение 10.9. Напишите функцию `contains(expression, variable)`, которая проверяет присутствие переменной `variable` в выражении `expression`.

Решение. Это легко сделать, проверив присутствие переменной в результате, возвращаемом функцией `distinct_variables`, тем не менее вот реализация с нуля:

```
def contains(exp, var):
    if isinstance(exp, Variable):
        return exp.symbol == var.symbol
    elif isinstance(exp, Number):
        return False
    elif isinstance(exp, Sum):
        return any([contains(e,var) for e in exp.exps])
    elif isinstance(exp, Product):
        return contains(exp.exp1,var) or contains(exp.exp2,var)
    elif isinstance(exp, Power):
        return contains(exp.base, var) or contains(exp.exponent, var)
    elif isinstance(exp, Apply):
        return contains(exp.argument, var)
    else:
        raise TypeError("Not a valid expression.")
```

Упражнение 10.10. Напишите функцию `distinct_functions`, которая принимает выражение и возвращает имена функций, например, `sin` или `ln`, встречающиеся в выражении.

Решение. Реализация очень похожа на создание функции `distinct_variables` в разделе 10.3.1:

```
def distinct_functions(exp):
    if isinstance(exp, Variable):
        return set()
    elif isinstance(exp, Number):
        return set()
    elif isinstance(exp, Sum):
        return set().union(*[distinct_functions(exp) for exp in exp.exps])
    elif isinstance(exp, Product):
        return distinct_functions(exp.exp1)
            .union(distinct_functions(exp.exp2))
    elif isinstance(exp, Power):
        return distinct_functions(exp.base)
            .union(distinct_functions(exp.exponent))
    elif isinstance(exp, Apply):
        return set([exp.function.name])
            .union(distinct_functions(exp.argument))
    else:
        raise TypeError("Not a valid expression.")
```

Упражнение 10.11. Напишите функцию `contains_sum`, которая принимает выражение и возвращает `True`, если оно содержит комбинатор `Sum`, и `False` — в противном случае.

Решение

```
def contains_sum(exp):
    if isinstance(exp, Variable):
        return False
    elif isinstance(exp, Number):
        return False
    elif isinstance(exp, Sum):
        return True
    elif isinstance(exp, Product):
        return contains_sum(exp.exp1) or contains_sum(exp.exp2)
    elif isinstance(exp, Power):
        return contains_sum(exp.base) or contains_sum(exp.exponent)
    elif isinstance(exp, Apply):
        return contains_sum(exp.argument)
    else:
        raise TypeError("Not a valid expression.")
```

Упражнение 10.12. Мини-проект. Напишите метод `__repr__` для классов `Expression`, который разборчиво и удобочитаемо отображал бы содержимое объекта в интерактивном сеансе.

Решение. Реализация приводится в блокноте Jupyter для главы 10. Обсуждение особенностей реализации `__repr__` и других специальных методов в Python можно найти в приложении Б.

Упражнение 10.13. Мини-проект. Если вы знаете, как представлять уравнения на языке LaTeX, то напишите метод `_repr_latex_` для классов `Expression`, который возвращает код на LaTeX, представляющий заданное выражение. После добавления этого метода вы сможете видеть красиво оформленные выражения в Jupyter.

```
In [41]:  1  Product(Power(Variable("x"),Number(2)),Apply(Function("sin"),Variable("y")))
Out[41]: x² sin(y)
```

Наличие метода `_repr_latex_` заставляет Jupyter использовать его для отображения уравнений в REPL.

Решение. Реализация приводится в блокноте Jupyter для главы 10.

Упражнение 10.14. Мини-проект. Напишите метод, генерирующий код на Python, представляющий выражение. Используйте функцию `eval`, чтобы преобразовать его в выполняемую функцию на Python. Сравните результат, возвращаемый этим методом, с результатом метода `evaluate`. Например, `Power(Variable("x"),Number(2))` представляет выражение x^2. Для этого выражения ваш метод должен сгенерировать код `x ** 2`. Затем передайте этот код функции `eval`, чтобы выполнить его, и сравните получившийся результат с результатом метода `evaluate`.

Решение. Реализация приводится в блокноте Jupyter для главы 10. Закончив работу над методом, вы должны иметь возможность выполнить такой код:

```
>>> Power(Variable("x"),Number(2))._python_expr()
'(x) ** (2)'
>>> Power(Variable("x"),Number(2)).python_function(x=3)
9
```

10.4. ПОИСК ПРОИЗВОДНОЙ ФУНКЦИИ

Это может показаться неочевидным, но зачастую существует четкая алгебраическая формула производной функции. Например, производная функции $f(x) = x^3$, дающая мгновенную скорость изменения f в любой точке x, определяется выражением $f'(x) = 3x^2$. Зная такую формулу, можно получить точный результат, такой как $f'(2) = 12$, без вычислительных проблем, связанных с использованием коротких секущих прямых.

В средней школе затрачивается довольно много времени на изучение формул производных и их поиск. Это простая задача, не требующая особого творчества, и довольно утомительная. Вот почему мы лишь кратко рассмотрим правила, а затем сосредоточимся на том, чтобы переложить всю остальную работу на Python.

10.4.1. Производные степеней

Найти производную линейной функции вида $f(x) = mx + b$ вы легко можете и без знания матанализа. Наклон любой секущей для этой линии независимо от длины секущей всегда совпадает с наклоном линии m, следовательно, $f'(x)$ не зависит от x. В частности, можно сказать, что $f'(x) = m$, потому что линейная функция $f(x)$ изменяется с постоянной скоростью по отношению к ее параметру x, соответственно, ее производная — константа. Кроме того, постоянный

член b не влияет на наклон линии, поэтому он отсутствует в формуле производной (рис. 10.12).

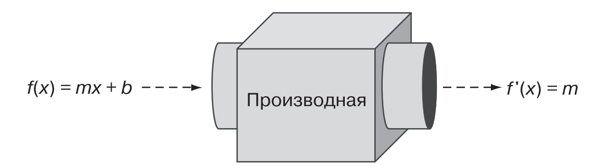


Рис. 10.12. Производная линейной функции — константа

Оказывается, производная квадратичной функции — это линейная функция. Например, $q(x) = x^2$ имеет производную $q'(x) = 2x$. Это станет видно, если построить график $q(x)$. Наклон $q(x)$ начинается с отрицательного значения, увеличивается и, наконец, становится положительным для $x \geq 0$. Функция $q'(x) = 2x$ согласуется с этим качественным описанием.

Другой наглядный пример: я уже показывал вам, что x^3 имеет производную $3x^2$. Все это — частные случаи общего правила, гласящего: производной степенной функции $f(x)$ является также степенная функция, но с показателем степени *на единицу меньше*. В частности, на рис. 10.13 показана производная функции вида ax^n, которая имеет вид nax^{n-1}.

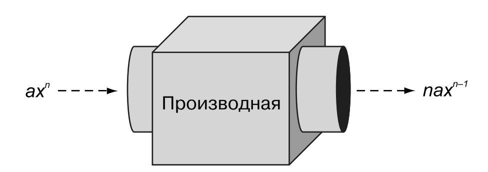


Рис. 10.13. Производной степенной функции $f(x)$ является степенная функция с показателем степени на единицу меньше

Разберем это правило на конкретном примере. Функция $g(x) = 5x^4$ — это функция вида ax^n с $a = 5$ и $n = 4$. Соответствующая ей производная определяется формулой nax^{n-1}, то есть $4 \cdot 5 \cdot x^{4-1} = 20x^3$. Так же как любую другую производную, которую мы рассмотрели в этой главе, вы можете перепроверить этот пример, нарисовав график производной на фоне результата, полученного численным методом с помощью функции производной из главы 9. Графики должны точно совпадать.

Линейная функция, такая как $f(x) = mx$, тоже является степенной функцией $f(x) = mx^1$. К ней тоже применимо степенное правило: mx^1 имеет производную $1 \cdot mx^0$, поскольку $x^0 = 1$. С точки зрения геометрии добавление константы b не меняет производную — константа лишь смещает график функции вверх или вниз, но не меняет его наклона.

10.4.2. Производные преобразованных функций

Добавление константы в функцию не меняет ее производной. Например, производная для x^{100} равна $100x^{99}$, производная $x^{100} - \pi$ также равна $100x^{99}$. Но некоторые преобразования, применяемые к функции, *действительно* меняют производную. Например, если поставить перед функцией знак «минус», то ее график перевернется, равно как и график любой ее секущей. Если наклон секущей равен m до переворота, то после переворота он станет равным $-m$, значение x такое же, как и раньше, но $y = f(x)$ теперь изменяется в противоположном направлении (рис. 10.14).

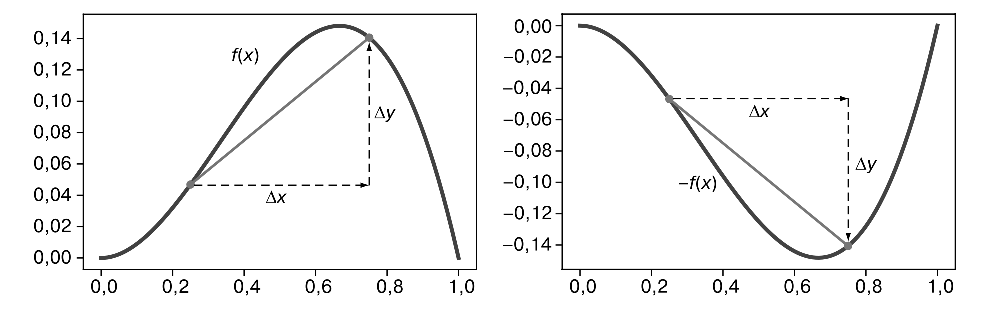


Рис. 10.14. Секущие для $f(x)$ и $-f(x)$ на том же интервале x имеют противоположный наклон

Поскольку производные определяются наклоном секущих, производная отрицательной функции $-f(x)$ равна отрицательной производной $-f'(x)$. Это согласуется с формулой, которую мы уже видели: если $f(x) = -5x^2$, то $a = -5$ и $f'(x) = -10x$ (сравните с функцией $5x^2$, производная которой имеет вид $+10x$). Иначе говоря, если функцию умножить на -1, то ее производная также умножается на -1.

То же верно для любой константы. Если $f(x)$ умножить на 4, чтобы получить $4f(x)$, то, как показано на рис. 10.15, график этой новой функции будет в четыре раза круче в каждой точке и, следовательно, ее производная будет равна $4f'(x)$.

Это согласуется с правилом взятия производных степенных функций, которое я показал ранее. Мы знаем, что производная для x^2 равна $2x$, а также что производная для $10x^2$ равна $20x$, производная $-3x^2$ равна $-6x$ и т. д. Мы пока не рассмотрели эту особенность, но если я скажу вам, что производная от $\sin(x)$ равна $\cos(x)$, то вы сразу поймете, что производная для $1{,}5 \cdot \sin(x)$ равна $1{,}5 \cdot \cos(x)$.

Последнее важное преобразование — это сложение двух функций. Если посмотреть на график $f(x) + g(x)$ для пары функций f и g, изображенный на рис. 10.16, то можно заметить, что приращение по вертикали для любой секущей будет равно сумме приращений по вертикали f и g на этом интервале.

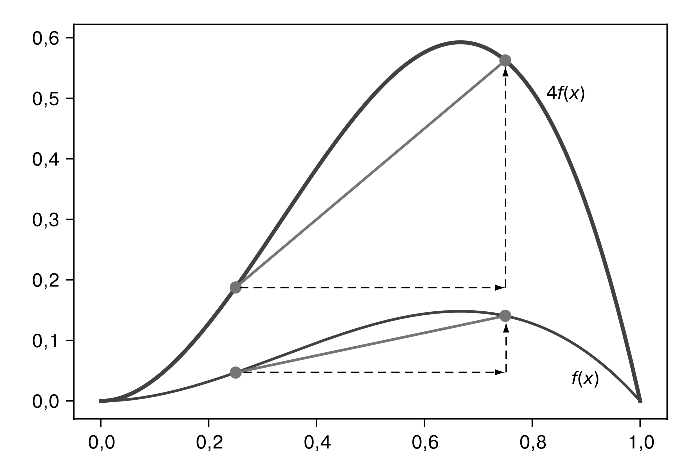


Рис. 10.15. Умножение функции на 4 делает каждую секущую линию в четыре раза круче

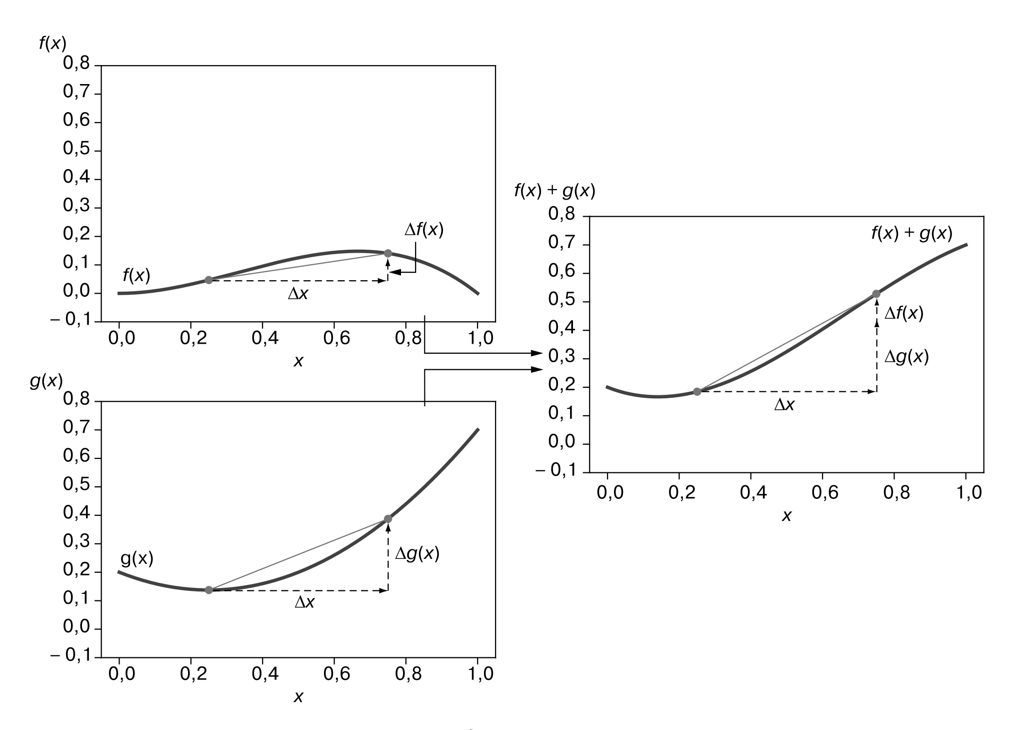


Рис. 10.16. Вертикальное приращение *f*(*x*) + *g*(*x*) на некотором интервале *x* — это сумма вертикальных приращений *f*(*x*) и *g*(*x*) на этом интервале

Работая с формулами, мы можем взять производную каждого члена в сумме независимо. Зная, что производная для x^2 равна $2x$ и производная для x^3 равна $3x^2$,

мы сможем вывести производную для $x^2 + x^3$, которая равна $2x + 3x^2$. Это правило более точно описывает причину того, почему производная для $mx + b$ равна m: производные слагаемых равны m и 0 соответственно, поэтому производная всей формулы $m + 0 = m$.

10.4.3. Производные некоторых специальных функций

Существует множество функций, которые нельзя записать в виде ax^n или даже в виде суммы этих форм, например, тригонометрические функции, экспоненциальные функции и логарифмы. Их необходимо рассматривать отдельно. На занятиях по математическому анализу рассказывают, как вычислять производные этих функций, но это объяснение выходит за рамки книги. Моя цель — показать, как использовать производные, чтобы, встретив их, вы смогли решить стоящую перед вами задачу. С этой целью я дам краткий список некоторых других важных правил взятия производных (табл. 10.1).

Таблица 10.1. Некоторые основные производные

Функция	Формула	Производная
Синус	$\sin(x)$	$\cos(x)$
Косинус	$\cos(x)$	$-\sin(x)$
Экспонента	e^x	e^x
Экспонента (с любым основанием)	a^x	$\ln(a) \cdot a^x$
Натуральный логарифм	$\ln(x)$	$1/x$
Логарифм (с любым основанием)	$\log_a x$	$\frac{1}{\ln(a) \cdot x}$

Вы можете использовать эту таблицу вместе с предыдущими правилами для вычисления производных сложных функций. Например, пусть $f(x) = 6x + 2\sin(x) + 5e^x$. Производная первого слагаемого равна 6 по правилу для степенных функций, рассмотренных в разделе 10.4.1. Второе слагаемое содержит функцию $\sin(x)$, производная которой равна $\cos(x)$, а двойка удваивает результат, давая нам $2\cos(x)$. Наконец, производная функции e^x равна сама себе (исключительный случай!), поэтому производная для $5e^x$ равна $5e^x$. Общая производная суммы $f'(x) = 6 + 2\cos(x) + 5e^x$.

Но будьте осторожны и не ограничивайтесь применением *только* тех правил, которые мы рассмотрели к настоящему моменту, для степенных функций (раздел 10.4.1) из табл. 10.1, а также для сумм и произведений на скаляр. Если встретите функцию $g(x) = \sin(\sin(x))$, у вас может возникнуть соблазн написать $g'(x) = \cos(\cos(x))$, заменив оба вхождения синуса в производной. Но это неправильно! Будет также ошибкой заключить, что производная произведения

$e^x \cos(x)$ равна $-e^x \sin(x)$. Когда функции комбинируются иными способами, отличными от сложения и вычитания, следует использовать другие правила для вычисления их производных.

10.4.4. Производные произведений и сложных функций

Рассмотрим произведение $f(x) = x^2 \sin(x)$. Эту функцию можно записать в виде произведения двух других функций: $f(x) = g(x) \cdot h(x)$, где $g(x) = x^2$ и $h(x) = \sin(x)$. Как я только что предупредил, здесь $f'(x)$ не равно $g'(x) \cdot h'(x)$. К счастью, есть еще одна верная формула, и называется она *правилом произведения*.

Правило произведения: если $f(x)$ можно записать как произведение двух других функций, g и h, например $f(x) = g(x) \cdot h(x)$, то производная $f(x)$ определяется так:

$$f'(x) = g'(x) \cdot h(x) + g(x) \cdot h'(x).$$

Попробуем применить это правило к $f(x) = x^2 \sin(x)$. В этом случае $g(x) = x^2$ и $h(x) = \sin(x)$, поэтому согласно правилам, приведенным ранее, $g'(x) = 2x$ и $h'(x) = \cos(x)$. Подставив их в формулу правила произведения $f'(x) = g'(x) \cdot h(x) + g(x) \cdot h'(x)$, получаем: $f'(x) = 2x \sin(x) + x^2 \cos(x)$. Вот и все!

Как видите, правило вычисления производной произведения функций согласуется с правилом вычисления производной степенной функции из раздела 10.4.1. Если переписать x^2 как произведение $x \cdot x$, то, согласно правилу произведения, производная равна $1 \cdot x + x \cdot 1 = 2x$.

Еще одно важное правило определяет порядок вычисления производных сложных (составных) функций, таких как $\ln(\cos(x))$. Эта функция имеет вид $f(x) = g(h(x))$, где $g(x) = \ln(x)$ и $h(x) = \cos(x)$. Мы не можем просто подставить производные на место функции и получить $-1/\sin(x)$ — на самом деле порядок вычислений немного сложнее. Формула производной функции вида $f(x) = g(h(x))$ называется *цепным правилом*.

Цепное правило: если $f(x)$ является композицией двух функций, то есть может быть записана в виде $f(x) = g(h(x))$, где g и h — некоторые функции, то производная для f вычисляется как

$$f'(x) = h'(x) \cdot g'(h(x)).$$

В нашем случае обе производные (см. табл. 10.1) имеют вид $g'(x) = 1/x$ и $h'(x) = -\sin(x)$. Подставив их в формулу цепного правила, получаем:

$$f'(x) = h'(x) \cdot g'(h(x)) = -\sin(x) \cdot \frac{1}{\cos(x)} = -\frac{\sin(x)}{\cos(x)}.$$

Вспомните также, что $\sin(x)/\cos(x) = \tan(x)$, поэтому формулу можно сократить и записать производную сложной функции $\ln(\cos(x))$ как $-\tan(x)$. В разделе с упражнениями вам будет предоставлено еще несколько возможностей попрактиковаться в применении цепного правила, а кроме того, можете обратиться к литературе по математическому анализу и найти другие примеры вычисления производных. Я не призываю верить мне на слово в отношении этих правил вычисления производных — попробуйте сами получить такой же результат, отыскав формулу производной или воспользовавшись функцией производной из главы 9. В следующем разделе я покажу, как преобразовать правила вычисления производных в программный код.

10.4.5. Упражнения

Упражнение 10.15. Покажите, что производная $f(x) = x^5$ действительно равна $f'(x) = 5x^4$, построив график производной, найденной численным способом (с помощью функции производной из главы 8), рядом с графиком символьной производной $f'(x) = 5x^4$.

Решение

```
def p(x):
    return x**5
plot_function(derivative(p), 0, 1)
plot_function(lambda x: 5*x**4, 0, 1)
```

Два графика совпадают.

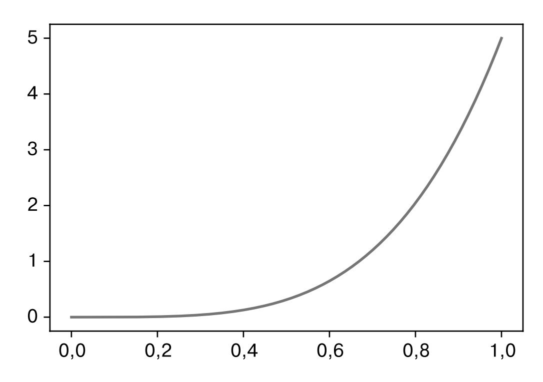


График функции $5x^4$ (вычисленный) и график производной от x^5

Упражнение 10.16. Мини-проект. Снова поразмыслим о функциях одной переменной как о векторном пространстве, как делали в главе 6. Объясните, почему правила вычисления производных предполагают, что производная является линейным преобразованием этого векторного пространства. (Ограничьтесь только функциями, имеющими производные на всем своем протяжении.)

Решение. Функции f и g как векторы можно складывать и умножать на скаляры. Напомню, что $(f + g)(x) = f(x) + g(x)$ и $(c \cdot f)(x) = c \cdot f(x)$. *Линейное преобразование* — это преобразование, сохраняющее векторные суммы и результаты умножения на скаляр.

Если записать производную как функцию D, то ее можно представить как функцию, которая принимает некоторую функцию и возвращает ее производную. Например, $Df = f'$. Производная суммы двух функций есть сумма производных:

$$D(f + g) = Df + Dg.$$

Производная произведения функции на число c в c раз больше производной исходной функции:

$$D(c \cdot f) = c \cdot Df.$$

Эти два правила означают, что D — это линейное преобразование. Отметим, в частности, что производная линейной комбинации функций есть такая же линейная комбинация их производных:

$$D(a \cdot f + b \cdot g) = a \cdot Df + b \cdot Dg.$$

Упражнение 10.17. Мини-проект. Найдите формулу производной частного $f(x)/g(x)$.

Подсказка. Используйте тот факт, что

$$\frac{f(x)}{g(x)} = f(x) \cdot \frac{1}{g(x)} = f(x) \cdot g(x)^{-1}.$$

Правило определения производных степенных функций выполняется и для отрицательных показателей: например, x^{-1} имеет производную $-x^{-2} = -1/x^2$.

Решение. Согласно цепному правилу производная функции $g(x)^{-1}$ равна $-g(x)^{-2} \cdot g'(x)$, или

$$-\frac{g'(x)}{g(x)^2}.$$

Соответственно, производная частного $f(x)/g(x)$ равна производной произведения $f(x) \cdot g(x)^{-1}$, которая определяется правилом вычисления производной произведения функций:

$$f'(x)g(x)^{-1} - \frac{g'(x)}{g(x)^2}f(x) = \frac{f'(x)}{g(x)} - \frac{f(x)g'(x)}{g(x)^2}.$$

Умножив уменьшаемое на $g(x)/g(x)$, приводим оба члена к общему знаменателю и получаем возможность выполнить вычитание:

$$\frac{f'(x)}{g(x)} - \frac{f(x)g'(x)}{g(x)^2} = \frac{f'(x)g(x)}{g(x)^2} - \frac{f(x)g'(x)}{g(x)^2} = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}.$$

Упражнение 10.18. Вычислите производную для произведения функций $\sin(x) \cdot \cos(x) \cdot \ln(x)$.

Решение. Здесь два произведения, и, к счастью, правило произведения можно применять в любом порядке — результат от этого не изменится. Производная произведения $\sin(x) \cdot \cos(x)$ равна $\sin(x) \cdot (-\sin(x)) + \cos(x) \cdot \cos(x) = \cos^2(x) - \sin^2(x)$. Производная $\ln(x)$ равна $1/x$, поэтому, согласно правилу произведения, производная всего произведения равна

$$\ln(x)\left(\cos(x)^2 - \sin(x)^2\right) + \frac{\sin(x)\cos(x)}{x}.$$

Упражнение 10.19. Предположим, нам известны производные трех функций, f, g и h, которые записываются как f', g' и h'. Чему равна производная $f(g(h(x)))$ по x?

Решение. Здесь требуется дважды применить цепное правило. Один член равен $f'(g(h(x)))$, но его нужно умножить на производную от $g(h(x))$. Она равна произведению $g'(h(x))$ и производной внутренней функции $h(x)$. Поскольку производная от $g(h(x))$ равна $h'(x) \cdot g'(h(x))$, то производная для $f(g(h(x)))$ равна $f'(x) \cdot g'(h(x)) \cdot f'(g(h(x)))$.

10.5. АВТОМАТИЧЕСКОЕ ВЗЯТИЕ ПРОИЗВОДНОЙ

Я показал вам лишь несколько правил взятия производных, тем не менее теперь вы сможете справиться практически с любой функцией. Если функция состоит из сумм, произведений, степеней и композиций функций, а также тригонометрических и экспоненциальных функций, вы сможете вычислить ее производную с помощью цепного правила, правила произведения и т. д.

Этот подход очень похож на использованный нами для построения алгебраических выражений в виде структур данных на Python. Несмотря на бесконечное число возможностей, все они формируются из одного и того же набора строительных блоков и предопределенных способов сборки. Чтобы реализовать автоматическое взятие производной, нужно предусмотреть все возможные случаи репрезентативного выражения элементов и комбинаторов и применить соответствующие правила для получения их производных. Конечным результатом является функция на Python, принимающая одно выражение и возвращающая другое — его производную.

10.5.1. Реализация метода вычисления производной для выражений

И снова реализуем функцию вычисления производной как метод в каждом из классов `Expression`. Чтобы гарантировать наличие этого метода в каждом подклассе, добавим абстрактный метод в абстрактный базовый класс:

```
class Expression(ABC):
    ...
    @abstractmethod
    def derivative(self,var):
        pass
```

Метод должен принимать параметр `var`, определяющий переменную, по которой берется производная. Например, производная от $f(y) = y^2$ будет браться по переменной y. В качестве более сложного примера мы рассматривали такие выражения, как ax^n, где a и n представляют константы и только x — переменная. С этой точки зрения производная равна nax^{n-1}. Но если эту функцию рассматривать как функцию от a, например, $f(a) = ax^n$, то производная будет равна x^{n-1} — константе в степени, равной константе. Иной результат получится, если будем рассматривать ее как функцию от n: если $f(n) = ax^n$, то $f'(n) = a \ln(n)\, x^n$. Чтобы избежать путаницы, далее будем рассматривать все выражения как функции переменной x.

Как обычно, самыми простыми примерами являются элементы — объекты `Number` и `Variable`. Производная числа всегда равна 0 независимо от переданной переменной:

```
class Number(Expression):
    ...
    def derivative(self,var):
        return Number(0)
```

Для функции $f(x) = x$ производная $f'(x) = 1$ — это наклон линии. Взятие производной от $f(x) = c$ даст 0, потому что c — это константа, а не аргумент функции f. По этой причине производная по переменной равна 1, только если это переменная, по которой берется производная, в противном случае производная равна 0:

```
class Variable(Expression):
    ...
    def derivative(self, var):
        if self.symbol == var.symbol:
            return Number(1)
        else:
            return Number(0)
```

Самый простой комбинатор с точки зрения взятия производной — `Sum`, производная функции `Sum` — это просто сумма производных ее членов:

```
class Sum(Expression):
    ...
    def derivative(self, var):
        return Sum(*[exp.derivative(var) for exp in self.exps])
```

После реализации этих методов мы уже можем вычислить производные в некоторых простых случаях. Например, выражение `Sum(Variable("x"),Variable("c"),Number(1))` представляет выражение $x + c + 1$. Рассматривая его как функцию от x, можно вычислить производную по отношению к x:

```
>>> Sum(Variable("x"),Variable("c"),Number(1)).derivative(Variable("x"))
Sum(Number(1),Number(0),Number(0))
```

Этот пример дает верную производную $1 + 0 + 0$ для $x + c + 1$, которая фактически равна 1. Мы получили не самое наглядное представление результата, но по крайней мере поняли его.

Я предлагаю в качестве самостоятельного упражнения написать упрощающий метод, который сокращает лишние члены производной, такие как нулевые слагаемые. Мы могли бы добавить логику упрощения выражений при вычислении производных, но сейчас лучше не смешивать стоящие перед нами задачи и сосредоточиться на вычислении производной. А теперь рассмотрим остальные комбинаторы.

10.5.2. Реализация правила произведения и цепного правила

Произведение — самый простой из оставшихся комбинаторов для реализации вычисления производной. Производная произведения, состоящего из двух выражений, определяется в терминах этих выражений и их производных. Как вы наверняка помните, производная произведения $g(x) \cdot h(x)$ — это $g'(x) \cdot h(x) + g(x) \cdot h'(x)$. Это правило можно выразить в виде программного кода, возвращающего результат как сумму двух произведений:

```
class Product(Expression):
    ...
    def derivative(self,var):
        return Sum(
            Product(self.exp1.derivative(var), self.exp2),
            Product(self.exp1, self.exp2.derivative(var)))
```

И снова мы получаем верные, хотя и не упрощенные результаты. Например, производная выражения cx по x:

```
>>> Product(Variable("c"),Variable("x")).derivative(Variable("x"))
Sum(Product(Number(0),Variable("x")),Product(Variable("c"),Number(1)))
```

Этот результат представляет выражение производной $0 \cdot x + c \cdot 1$, то есть c.

Теперь наша реализация обрабатывает комбинаторы `Sum` и `Product`, поэтому далее рассмотрим обработку комбинатора `Apply`. Чтобы вычислить производную функции, такой как $\sin(x^2)$, нужно не только получить производную синуса, но и применить цепное правило к x^2 в круглых скобках.

Для начала перечислим производные некоторых специальных функций с помощью переменной-заполнителя, имя которой, `'placeholder variable'`, вряд ли можно спутать с именем любой переменной, используемой на практике. Производные будут храниться в виде словаря, ключами в котором являются имена функций, а значениями — выражения, определяющие их производные:

```
_var = Variable('placeholder variable')

_derivatives = {
    "sin": Apply(Function("cos"), _var),
    "cos": Product(Number(-1), Apply(Function("sin"), _var)),
    "ln": Quotient(Number(1), _var)
}
```

Переменная-заполнитель определяется так, что ее нельзя спутать с каким-либо символом (например, x или y), который мог бы использоваться на практике

Запись, указывающая, что производная синуса является косинусом, причем производная представлена выражением с использованием переменной-заполнителя

Следующий шаг — добавление метода `derivative` в класс `Apply`, отыскивающий производную в словаре `_derivatives` и применяющий цепное правило. Напомню,

что производная функции $g(h(x))$ равна $h'(x) \cdot g'(h(x))$. Если, например, дана функция $\sin(x^2)$, то $g(x) = \sin(x)$ и $h(x) = x^2$. Сначала метод обращается к словарю, чтобы получить производную для sin, и получает cos со значением-заполнителем. Затем нужно подставить $h(x) = x^2$ на место заполнителя, чтобы получить член $g'(h(x))$, определяемый цепным правилом. Для этого потребуется функция подстановки, которая заменит все экземпляры переменной выражением (я предлагаю написать эту функцию `substitute` в качестве самостоятельного упражнения). Ее реализацию можно найти в примерах с исходным кодом. Вот как выглядит метод `derivative` для `Apply`:

```
class Apply(Expression):
    ...
    def derivative(self, var):
        return Product(
            self.argument.derivative(var),
            _derivatives[self.function.name].substitute(_var, self.argument))
```

Возвращает h′(x) для формулы цепного правила h′(x) · g′(h(x))

Возвращает g′(h(x)) для формулы цепного правила; словарь _derivatives используется для поиска g′, а h(x) просто подставляется

Для $\sin(x^2)$, например, получим:

```
>>> Apply(Function("sin"),Power(Variable("x"),Number(2))).derivative(x)
Product(Product(Number(2),Power(Variable("x"),Number(1))),Apply(Function("cos
"),Power(Variable("x"),Number(2))))
```

Этот результат является буквальным представлением выражения $(2x^1) \cdot \cos(x^2)$ — верным результатом применения цепного правила.

10.5.3. Реализация степенного правила

Последний тип выражений, обработку которого мы должны предусмотреть, — это комбинатор возведения в степень. На самом деле в методе `derivative` класса `Power` нужно реализовать три правила вычисления производной. Первое правило, которое я назвал степенным, гласит, что выражение x^n имеет производную nx^{n-1}, когда n — константа. Второе — это производная функции a^x, где основание a является константой, а показатель степени — переменной. Эта функция имеет производную $\ln(a) \cdot a^x$.

Наконец, третье — цепное правило, потому что выражение с переменной может присутствовать в основании или показателе степени, например, $\sin(x)^8$ или $15^{\cos(x)}$. Есть еще один случай, когда выражение с переменной используется *и* в основании, *и* в показателе степени, например, x^x или $\ln(x)^{\sin(x)}$. За все годы работы с производными я ни разу не видел примеров этого третьего случая, поэтому опущу его и просто сгенерирую исключение, если он вдруг встретится.

Поскольку все типы выражений — x^n, $g(x)^n$, a^x и $a^{g(x)}$ —представлены на языке Python в виде экземпляра `Power(expression1, expression2)`, мы должны выполнить некоторые проверки, чтобы выяснить, какое правило применить. Если показатель степени — число, то используем правило x^n, если основание — число, то правило a^x. В обоих случаях по умолчанию будет применяться цепное правило. В конце концов, x^n — это частный случай $f(x)^n$, где $f(x) = x$. Вот как выглядит сама реализация:

```
class Power(Expression):
    ...
    def derivative(self,var):
        if isinstance(self.exponent, Number):
            power_rule = Product(
                    Number(self.exponent.number),
                    Power(self.base, Number(self.exponent.number - 1)))
            return Product(self.base.derivative(var),power_rule)
        elif isinstance(self.base, Number):
            exponential_rule = Product(
                Apply(Function("ln"),
                Number(self.base.number)
            ),
            self)
            return Product(
                self.exponent.derivative(var),
                exponential_rule)
        else:
            raise Exception(
            "can't take derivative of power {}".format(self.display()))
```

Если показатель степени — число, то использовать степенное правило

Производная для $f(x)^n$ равна $f'(x) \cdot nf(x)^{n-1}$, поэтому здесь выполняется умножение множимого $f'(x)$ по цепному правилу

Если основание — число, то использовать экспоненциальное правило

Умножить на коэффициент $f'(x)$ при попытке взять производную от $af(x)$, согласно цепному правилу

В последнем случае, когда ни основание, ни показатель степени не являются числом, возникает ошибка. Теперь, реализовав этот последний метод, мы получили законченный калькулятор производных! Он может обрабатывать почти любые выражения, построенные из наших элементов и комбинаторов. Если применить его к исходному выражению $(3x^2 + x) \sin(x)$, то он вернет перегруженный деталями, но все-таки верный результат:

$$(0 \cdot x^2 + 3 \cdot 1 \cdot 2 \cdot x^1 + 1) \cdot \sin(x) + (3 \cdot x^2 + x) \cdot 1 \cdot \cos(x).$$

Это выражение упрощается до $(6x + 1) \sin(x) + (3x^2 + x) \cos(x)$ и является верным результатом применения правила произведения и степенного правила. В начале этой главы вы уже знали, как использовать Python для выполнения арифметических операций, а теперь знаете, как заставить Python выполнять еще и алгебраические операции. Теперь можете с полной уверенностью сказать, что умеете выполнять вычисления и на Python! В заключительном разделе я немного расскажу о символьном вычислении интегралов на Python с помощью готовой библиотеки SymPy.

10.5.4. Упражнения

Упражнение 10.20. Наша реализация уже обрабатывает случай, когда один из множителей в произведении — константа, то есть когда имеется произведение в форме $c \cdot f(x)$ или $f(x) \cdot c$. В любом случае производная равна $c \cdot f'(x)$. Нам не нужен второй член, вытекающий из правила произведения, то есть $f(x) \cdot 0 = 0$. Измените код, принимающий производную произведения, так, чтобы он напрямую обрабатывал этот случай вместо включения нулевого члена.

Решение. Для этого можно проверить, является ли выражение в произведении экземпляром класса `Number`. Более общее решение состоит в том, чтобы посмотреть, содержит ли какой-нибудь член произведения переменную, по которой берется производная. Например, производная от $(3 + \sin(5a))\, f(x)$ по x не требует применения правила произведения, потому что первый член не содержит переменной x. Следовательно, его производная по x равна 0. Выполнить эту проверку можно с помощью функции `contains(expression, variable)` из упражнения 10.9:

```
class Product(Expression):
    ...
    def derivative(self,var):
        if not contains(self.exp1, var):
            return Product(self.exp1, self.exp2.derivative(var))
        elif not contains(self.exp2, var):
            return Product(self.exp1.derivative(var), self.exp2)
        else:
            return Sum(
                Product(self.exp1.derivative(var), self.exp2),
                Product(self.exp1, self.exp2.derivative(var)))
```


Упражнение 10.21. Добавьте функцию вычисления квадратного корня в словарь известных функций и автоматически вычислите ее производную.

Подсказка. Квадратный корень из x — это возведение в степень $x^{1/2}$.

Решение. Согласно степенному правилу производная квадратного корня из x по x равна $1/2 \cdot x^{-1/2}$, что можно записать как

$$\frac{1}{2} \cdot \frac{1}{x^{1/2}} = \frac{1}{2\sqrt{x}}.$$

Вот как можно выразить эту формулу производной в программном коде:

```
_function_bindings = {
    ...
    "sqrt": math.sqrt
}

_derivatives = {
    ...
    "sqrt": Quotient(Number(1), Product(Number(2), Apply(Function("sqrt"),
_var)))
}
```

10.6. СИМВОЛЬНОЕ ИНТЕГРИРОВАНИЕ ФУНКЦИЙ

Другая операция матанализа, с которой мы познакомились в двух предыдущих главах, — это интегрирование. Если производная возвращает функцию, описывающую скорость изменения некоторой заданной функции, то интеграл выполняет обратную операцию — восстанавливает функцию по скорости ее изменения.

10.6.1. Интегралы как первообразные

Например, производная функции $y = x^2$ говорит нам, что мгновенная скорость изменения y по отношению к x равна $2x$. Если изначально имеется функция $2x$, то неопределенный интеграл отвечает на вопрос: какая функция от x имеет мгновенную скорость изменения, равную $2x$? По этой причине неопределенные интегралы также называют *первообразными*.

Один из возможных результатов неопределенного интеграла от $2x$ по x — это x^2, но есть и другие варианты — $x^2 - 6$ или $x^2 + \pi$. Поскольку производная любой константы равна 0, неопределенный интеграл не имеет однозначного результата. Имейте в виду, что даже зная, какую скорость показывает спидометр автомобиля в течение всей поездки, по одним лишь его показаниям нельзя сказать, где началась поездка и где она закончилась. По этой причине мы говорим, что x^2 — это *обобщенная* первообразная для $2x$, но не *точная* первообразная.

Если требуется выяснить *точную* первообразную, или *точное* значение неопределенного интеграла, мы должны добавить неуказанную константу, записав что-то вроде $x^2 + C$, где C называется константой интегрирования и пользуется дурной славой у студентов, изучающих математический анализ. Она кажется формальностью, но это важная формальность, и большинство преподавателей снижают оценку, если учащиеся забывают об этом.

Некоторые интегралы очевидны для имеющих некоторый опыт работы с производными. Например, интеграл от cos (x) по x записывается так:

$$\int \cos(x)dx.$$

И его результатом будет sin (x) + C, потому что для любой константы C производная от sin (x) + C равна cos (x). Если вы еще не забыли степенное правило, то, вероятно, сможете решить интеграл

$$\int 3x^2 dx.$$

Выражение $3x^2$ — это то, что получится в результате применения степенного правила к x^3, поэтому интеграл

$$\int 3x^2 dx = x^3 + C.$$

Есть более сложные интегралы, такие как

$$\int \tan(x)dx,$$

которые не имеют очевидного решения. Чтобы решить такой интеграл, нужно применить несколько правил взятия производной в обратном порядке. Выяснению способов решения таких непростых интегралов уделяется много времени в курсе математического анализа. Ситуация усугубляется еще и тем, что некоторые интегралы *невозможны*. Как известно, для функции:

$$f(x) = e^{x^2}$$

невозможно найти формулу неопределенного интеграла, по крайней мере, не придумывая новую функцию для ее представления. Но не буду мучить вас массой правил интеграции, лучше покажу, как использовать готовую библиотеку с функцией `integrate`, способной обрабатывать интегралы.

10.6.2. Введение в библиотеку SymPy

Библиотека SymPy (*Sym*bolic *Py*thon) — это библиотека для Python с открытым исходным кодом, предназначенная для поиска решений в символьной форме. Она определяет свои структуры данных для представления выражений, очень похожие на те, которые построили мы, а также перегруженные операторы, что делает эти выражения похожими на обычный код на Python. Вот пример кода, написанный с использованием SymPy (обратите внимание на то, насколько он похож на код, который мы писали ранее):

```
>>> from sympy import *
>>> from sympy.core.core import *
>>> Mul(Symbol('y'),Add(3,Symbol('x')))
y*(x + 3)
```

Конструкторы `Mul`, `Symbol` и `Add` заменяют конструкторы `Product`, `Variable` и `Sum`, но действуют похожим образом. SymPy также поощряет использование сокращений, например, код

```
>>> y = Symbol('y')
>>> x = Symbol('x')
>>> y*(3+x)
y*(x + 3)
```

создает эквивалентную структуру данных, представляющую выражение. Эта структура данных способна выполнять подстановку и находить производные:

```
>>> y*(3+x).subs(x,1)
4*y
>>> (x**2).diff(x)
2*x
```

Безусловно, SymPy — гораздо более надежная библиотека, чем та, которую написали мы в этой главе. И как видите, она автоматически упрощает выражения.

Я решил представить SymPy, чтобы показать ее мощную функцию символьной интеграции. Вот, например, как можно найти интеграл выражения $3x^2$:

```
>>> (3*x**2).integrate(x)
x**3
```

То есть эта функция сообщает, что

$$\int 3x^2 dx = x^3 + C.$$

В следующих нескольких главах мы продолжим применять производные и интегралы в работе.

10.6.3. Упражнения

Упражнение 10.22. Чему равен интеграл от $f(x) = 0$? Подтвердите ответ с помощью SymPy. Не забывайте при этом, что SymPy не включает в ответ константу интеграции.

Решение. Этот вопрос можно сформулировать иначе: какая функция имеет нулевую производную? Любая функция, дающая постоянное значение, на всем протяжении имеет нулевой наклон и, соответственно, нулевую производную. Интеграл

$$\int f(x) dx = \int 0\, dx = C.$$

Вызов конструктора `Integer(0)` из библиотеки SymPy дает число 0 в форме выражения, поэтому интеграл по переменной x равен

```
>>> Integer(0).integrate(x)
0
```

Ноль как функция — это одна из первообразных нуля. Добавляя константу интегрирования, получаем $0 + C$ или просто C, что соответствует нашему выводу. Любая функция, дающая постоянное значение, является первообразной нулевой функции.

Упражнение 10.23. Чему равен интеграл от $x \cos(x)$?

Подсказка. Посмотрите на производную $x \sin(x)$. Подтвердите свой ответ с помощью SymPy.

Решение. Начнем с подсказки — производная от $x \sin(x)$ равна $\sin(x) + x \cos(x)$ согласно правилу произведения. Это почти то, что нам нужно, но для дополнительного члена $\sin(x)$. Если бы у нас был член $-\sin(x)$, присутствующий в производной, он сократился бы из-за наличия этого дополнительного $\sin(x)$ и производная $\cos(x)$ была бы равна $-\sin(x)$. То есть производная от $x \sin(x) + \cos(x)$ равна $\sin(x) + x \cos(x) - \sin(x) = x \cos(x)$. Это и есть искомый результат, поэтому интеграл

$$\int x \cos(x) dx = x \sin(x) + \cos(x) + C.$$

Наш ответ совпадает с результатом, который дает SymPy:

```
>>> (x*cos(x)).integrate(x)
x*sin(x) + cos(x)
```

Такой способ восстановления производной как одного из членов произведения называется *интеграцией по частям*, и его любят преподаватели математического анализа во всем мире.

Упражнение 10.24. Чему равен интеграл от x^2? Подтвердите свой ответ с помощью SymPy.

Решение. Если $f'(x) = x^2$, то $f(x)$, вероятно, содержит член x^3, потому что согласно степенному правилу показатель степени уменьшается на единицу.

Производная x^3 равна $3x^2$, поэтому нам нужен дополнительный член, дающий треть этого результата, то есть $x^3/3$, имеющий производную x^2. Иначе говоря,

$$\int x^2 dx = \frac{x^3}{3} + C.$$

SymPy подтверждает это решение:

```
>>> (x**2).integrate(x)
x**3/3
```

КРАТКИЕ ИТОГИ ГЛАВЫ

- Моделирование алгебраических выражений в виде структур данных позволяет писать программы, отвечающие на многие вопросы о выражениях.
- Естественным способом моделирования алгебраического выражения в программном коде является *дерево*. Узлы дерева можно разделить на элементы (переменные и числа), являющиеся самостоятельными выражениями, и комбинаторы (суммы, произведения и т. д.), содержащие два или более выражений в качестве поддеревьев.
- Выполняя рекурсивный обход дерева выражений, можно ответить на вопросы о нем, например, какие переменные оно содержит. Можно также вычислить или упростить выражение либо перевести его на другой язык.
- Если известно выражение, определяющее функцию, то с помощью комплекса правил можно преобразовать его в выражение, представляющее производную этой функции. Среди них правило произведения и цепное правило, которые определяют, как должны браться производные произведений и композиций функций соответственно.
- Запрограммировав правила взятия производных, соответствующие каждому комбинатору в дереве выражений, можно получить функцию на Python, которая автоматически находит выражения, представляющие производные.
- SymPy — это надежная библиотека для работы с алгебраическими выражениями в коде на Python. В ней имеются функции упрощения, замены и вычисления производных. А также есть функция символьного интегрирования, которая сообщает формулу неопределенного интеграла функции.

11

Моделирование силовых полей

В этой главе

- ✓ Моделирование сил, таких как гравитация, с использованием скалярных и векторных полей.
- ✓ Вычисление векторов сил с помощью градиента.
- ✓ Получение градиента функции в Python.
- ✓ Добавление силы гравитации в игру с астероидами.
- ✓ Вычисление градиентов и работа с векторными полями более высоких измерений.

Только что во вселенной нашей игры с астероидами произошло катастрофическое событие — в центре игрового поля появилась черная дыра! В результате, как показано на рис. 11.1, космический корабль и все астероиды стали испытывать гравитационное притяжение к ней. Это делает игру еще более сложной, а также ставит перед нами новую математическую задачу — изучение и моделирование *силовых полей*.

Гравитация — знакомый пример силы, действующей на расстоянии, то есть чтобы почувствовать гравитационное притяжение, нет необходимости касаться объекта. Например, когда вы летите на самолете, то все еще можете нормально ходить по салону, потому что даже на высоте 10 000 м Земля притягивает вас к себе, вниз. Другие знакомые силы, действующие на расстоянии, — магнетизм

и статическое электричество. В физике источники такого рода сил представляются как магниты или статически заряженные воздушные шары, создающие вокруг себя невидимое силовое поле. В любом месте поля гравитационных сил Земли, называемого просто гравитационным полем, объект испытывает притяжение к Земле.



Рис. 11.1. Только не это! В центре поля появилась черная дыра

Наша главная задача в этой главе состоит в том, чтобы добавить модель гравитационного поля в игру с астероидами, закончив реализацию, мы познакомимся с математическим описанием в более общем виде, а именно с *векторными полями* — математическими функциями, моделирующими силовые поля. Векторные поля часто определяются результатом операции, называемой *градиентом*, которая является ключевым инструментом в примерах машинного обучения, которые мы рассмотрим в части III.

Математические выкладки и код в этой главе не особенно сложны, но здесь вам встретится много новых понятий, которые обязательно нужно освоить. Поэтому, чтобы вам было интереснее, я хочу изложить сюжетную канву главы, прежде чем мы углубимся в нее всерьез.

11.1. МОДЕЛИРОВАНИЕ ГРАВИТАЦИИ С ПОМОЩЬЮ ВЕКТОРНОГО ПОЛЯ

Векторное поле определяет конкретный вектор в каждой точке пространства. Гравитационное поле — это векторное поле, сообщающее, насколько сильно влияние гравитации и в каком направлении действует эта сила в любой заданной точке. Мы можем изобразить векторное поле как группу точек и нарисовав вектор из каждой из них в виде стрелки. Например, гравитационное поле, создаваемое черной дырой в нашей игре, можно изобразить, как показано на рис. 11.2.

Схема на рис. 11.2 согласуется с нашими интуитивными представлениями о гравитации: все стрелки указывают в направлении черной дыры, соответственно, любой объект, помещенный в эту область, будет притягиваться к ней. Чем ближе объект к черной дыре, тем сильнее на него действует гравитационное притяжение, поэтому стрелки длиннее.

Первое, что мы сделаем в этой главе, — смоделируем гравитационные поля в виде функций, принимающих точку в пространстве и сообщающих величину и направление силы, действующей на объект в этой точке. В программе на Python двухмерное векторное поле — это функция, принимающая двухмерный вектор,

представляющий точку, и возвращающая двухмерный вектор, представляющий силу в этой точке.

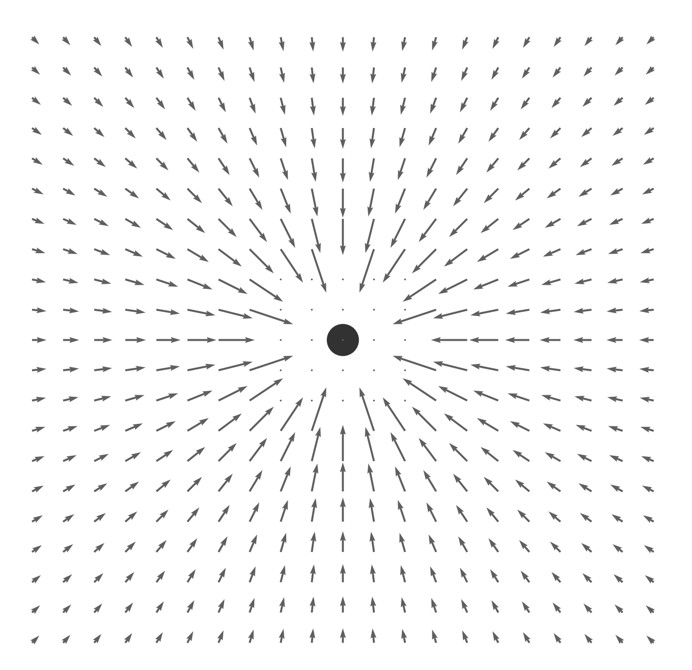


Рис. 11.2. Изображение гравитационного поля, создаваемого черной дырой в игре с астероидами

Написав эту функцию, мы используем ее, чтобы добавить гравитационное поле в игру с астероидами. Она скажет нам, какие гравитационные силы испытывают космический корабль и астероиды в зависимости от их местонахождения и какими должны быть скорость и направление ускорения. Реализовав воздействие ускорения, мы увидим, как объекты в игре с астероидами ускоряются в направлении черной дыры.

11.1.1. Моделирование гравитации с помощью функции потенциальной энергии

После модели гравитационного поля мы рассмотрим вторую, эквивалентную модель силы, действующей на расстоянии, называемую потенциальной энергией. Потенциальную энергию можно рассматривать как накопленную энергию, готовую к преобразованию в движение. Например, лук и стрелы изначально не имеют потенциальной энергии, но когда вы натягиваете тетиву, он приобретает потенциальную энергию. Когда тетива отпускается, эта энергия преобразуется в движение стрелы (рис. 11.3).

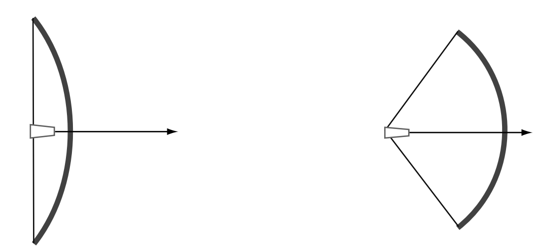


Рис. 11.3. Лук слева не имеет потенциальной энергии. Лук справа имеет значительный запас потенциальной энергии, которая готова привести стрелу в движение

Удаление космического корабля от черной дыры можно представить как натягивание воображаемого лука. Чем дальше удаляется космический корабль от черной дыры, тем большей потенциальной энергией он обладает и тем быстрее будет двигаться после освобождения. Мы смоделируем потенциальную энергию как еще одну функцию на Python, принимающую двухмерный вектор местоположения объекта в игровом мире и возвращающую число с оценкой его потенциальной энергии в этой точке. Когда каждой точке пространства присваивается некоторое число (вместо вектора), это называется *скалярным полем*.

Написав функцию потенциальной энергии, мы используем несколько визуализаций Matplotlib, чтобы посмотреть, как выглядит ее распределение в игровом пространстве. Один из важных примеров — *тепловая карта*, в которой применяются разные цвета и их оттенки, чтобы показать, как меняется значение скалярного поля в двухмерном пространстве (рис. 11.4).

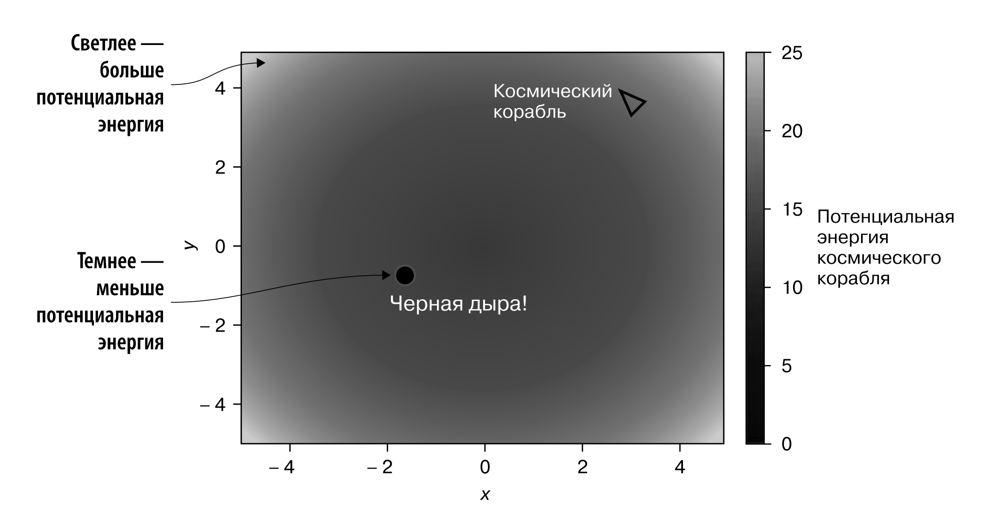


Рис. 11.4. Тепловая карта потенциальной энергии. Более светлые оттенки соответствуют областям с большей потенциальной энергией

Как показано на рисунке, при удалении от черной дыры цвета становятся светлее, а это означает, что потенциальная энергия увеличивается. Скалярное поле, представляющее потенциальную энергию, — это иная математическая модель, отличная от векторного поля, представляющего гравитацию, но оба они представляют одну и ту же физику. Кроме того, они математически связаны операцией, называемой *градиентом*.

Градиент скалярного поля — это векторное поле, сообщающее направление и величину наибольшего увеличения значений скалярного поля. В нашем примере потенциальная энергия увеличивается по мере удаления от черной дыры, поэтому градиент потенциальной энергии представляет собой векторное поле, направленное наружу в каждой точке. Если наложить векторное поле градиента на тепловую карту потенциальной энергии (рис. 11.5), то можно увидеть, что стрелки указывают в направлении увеличения потенциальной энергии.

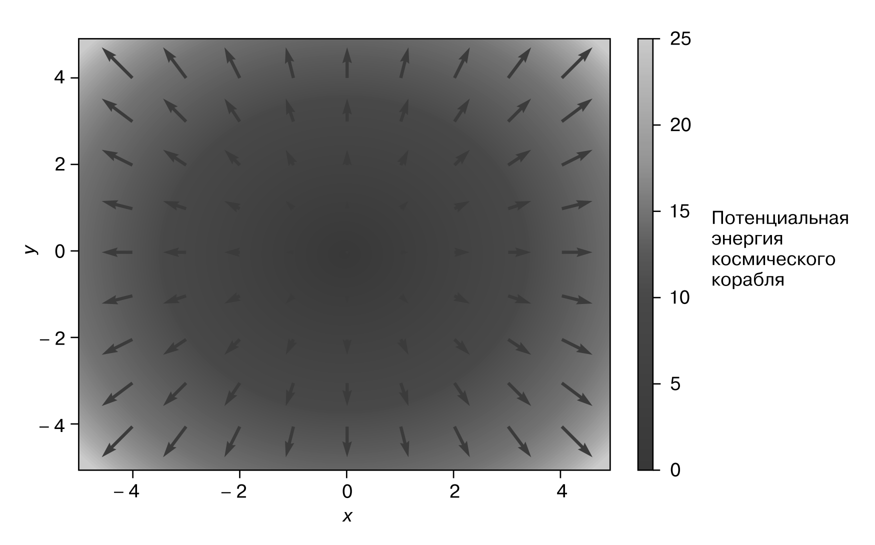


Рис. 11.5. Тепловая карта на основе функции потенциальной энергии с наложенным градиентом (векторным полем). Градиент указывает направление увеличения потенциальной энергии

Векторное поле градиента на рис. 11.5 похоже на гравитационное поле черной дыры, только стрелки указывают в противоположных направлениях, а величины меняются местами. Чтобы получить гравитационное поле из функции потенциальной энергии, нужно взять градиент, а затем поменять направления векторов силового поля, добавив знак «минус». В конце главы я покажу, как вычислить градиент скалярного поля с помощью производных, которые позволяют перейти от модели гравитации с потенциальной энергией к модели гравитации с силовым полем.

Теперь, получив представление о нашей цели, можно приступать к делу. Первое, что мы сделаем, — поближе познакомимся с векторными полями и посмотрим, как превратить их в функции на Python.

11.2. МОДЕЛИРОВАНИЕ ГРАВИТАЦИОННЫХ ПОЛЕЙ

Чтобы получить векторное поле, нужно каждой точке пространства присвоить вектор, например, вектор гравитационной силы. Далее мы будем рассматривать исключительно двухмерные векторные поля, в которых каждой точке двухмерного пространства присваивается двухмерный вектор. Для начала создадим конкретные представления векторных полей в виде функций на Python, принимающих и возвращающих двухмерные векторы. В примерах исходного кода вы найдете функцию `plot_vector_field`, которая принимает функцию векторного поля и рисует это поле, нанося на двухмерную плоскость множество точек с соответствующими двухмерными векторами.

Затем мы напишем код, добавляющий черную дыру в игру с астероидами. Черная дыра будет изображаться как простой черный круг и оказывать гравитационное воздействие на все объекты вокруг нее, как показано на рис. 11.6.

Для представления черной дыры мы реализуем класс `BlackHole`, определим соответствующее ему гравитационное поле как функцию, а затем дополним реализацию игрового цикла, добавив в него реакцию космического корабля и астероидов на воздействующие на них силы в соответствии с законами Ньютона.

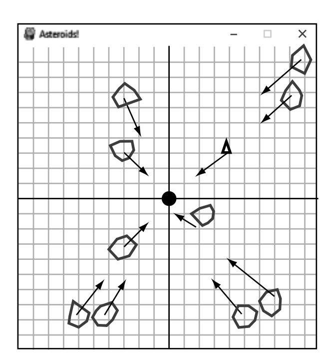


Рис. 11.6. Черная дыра в игре с астероидами — это черный круг, и каждый объект в игре испытывает гравитационное воздействие с ее стороны

11.2.1. Определение векторного поля

Ненадолго остановимся на основных обозначениях векторных полей. Векторное поле в двухмерной плоскости — это функция $\mathbf{F}(x, y)$, принимающая вектор, представленный двумя координатами, x и y. Она возвращает другой двухмерный вектор — значение векторного поля в точке (x, y). Жирный шрифт $\mathbf{F}$ означает, что возвращаемое значение является вектором и $\mathbf{F}$ — векторная функция. Когда мы говорим о векторных полях, то обычно интерпретируем входные данные как точки на плоскости, а выходные — как стрелки. На рис. 11.7 показана схема векторного поля $\mathbf{F}(x, y) = (-2y, x)$.

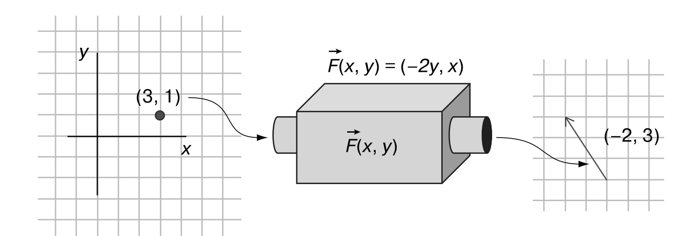


Рис. 11.7. Векторное поле **F**(*x*, *y*) = (–2*y*, *x*) принимает точку (3, 1) и возвращает стрелку (–2, 3)

Обычно выходной вектор рисуется в виде стрелки, начинающейся в точке на плоскости, которая играла роль входного вектора, так, что выходной вектор «прикрепляется» к входной точке (рис. 11.8).

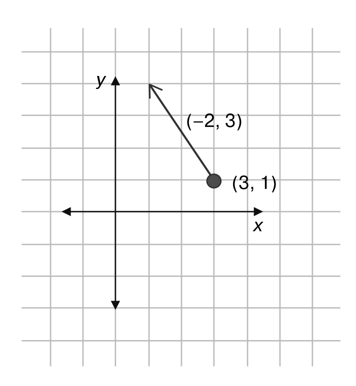


Рис. 11.8. Вектор (–2, 3) прикрепляется к точке (3, 1)

Рассчитав несколько значений **F**, можно изобразить векторное поле, нарисовав сразу несколько стрелок, прикрепленных к соответствующим точкам. На рис. 11.9 показаны еще три точки, (–2, 2), (–1, –2) и (–1, –2), с прикрепленными к ним стрелками, представляющими значения **F** в них. Результатами являются (–4, –2), (4, –1) и (4, 3) соответственно.

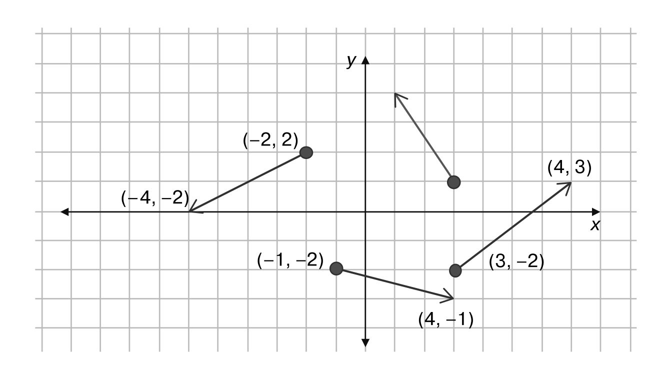


Рис. 11.9. Стрелки, прикрепленные к точкам, представляют большее количество значений векторного поля **F**(*x*, *y*) = (–2*y*, *x*)

Если нарисовать еще больше стрелок, они начнут перекрываться и рисунок станет неразборчивым. Чтобы избежать этого, длина векторов обычно уменьшается на постоянный коэффициент. Я включил в примеры исходного кода функцию-обертку `plot_vector_field`, использующую Matplotlib, и вы можете

применить ее, как показано далее, чтобы нарисовать векторное поле. Как показано на рис. 11.10, векторное поле $\mathbf{F}(x, y)$ циркулирует против часовой стрелки вокруг начала координат:

```
def f(x,y):
    return (-2*y, x)

plot_vector_field(f, -5,5,-5,5)
```

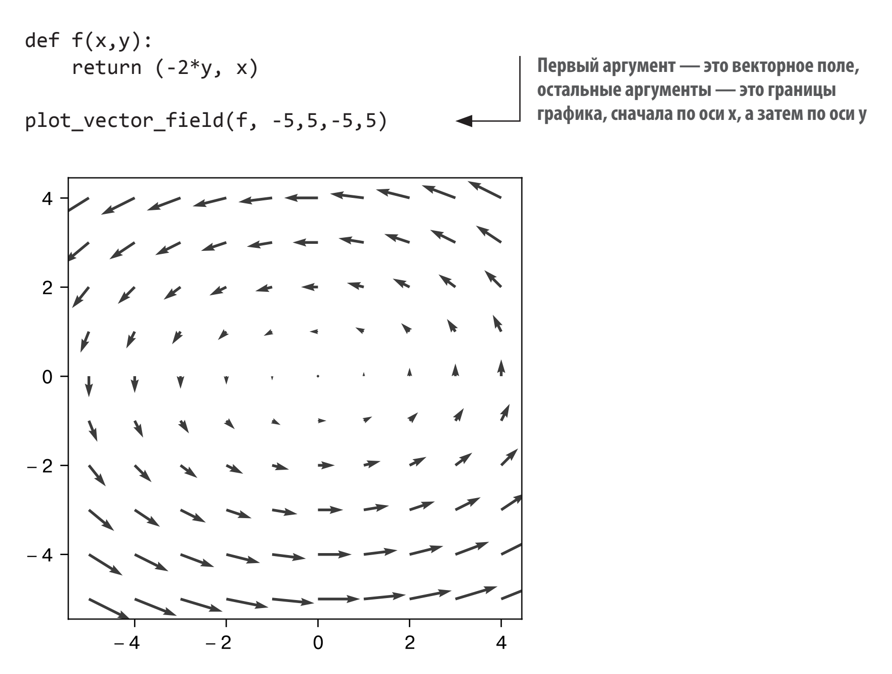


Рис. 11.10. График **F**(*x*, *y*), сгенерированный с помощью Matplotlib, в виде векторов, исходящих из точек (*x*, *y*)

Одно из значимых положений в физике заключается в особенностях моделирования различных сил в виде векторных полей. Следующий пример, который мы рассмотрим, — упрощенная модель гравитации.

11.2.2. Определение простого силового поля

Как нетрудно догадаться, сила гравитации усиливается по мере приближения к ее источнику. Несмотря на то что у Солнца гравитация сильнее, чем у Земли, мы с вами намного ближе к Земле, поэтому ощущаем только земную гравитацию. Для простоты не будем использовать реалистичное представление гравитационного поля, а представим его в виде векторного поля $\mathbf{F}(r) = -r$, которое равно $\mathbf{F}(x, y) = (-x, -y)$ на плоскости. Вот как это выглядит в коде (на рис. 11.11 показано, как это же поле изображается на графике):

```
def f(x,y):
    return (-x,-y)
plot_vector_field(f,-5,5,-5,5)
```

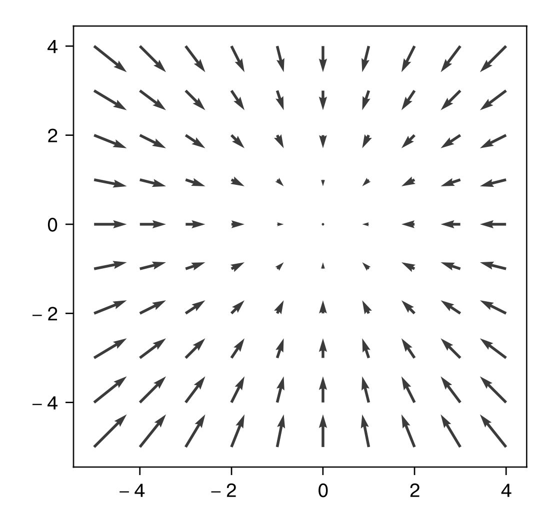


Рис. 11.11. Визуализация векторного поля **F**(*x*, *y*) = (–*x*, –*y*)

Это векторное поле похоже на гравитационное в том смысле, что во всех точках направлено к началу координат, но у него есть преимущество — действующая сила увеличивается по мере удаления. Это гарантирует, что моделируемый объект не сможет достичь космической скорости и исчезнуть из поля зрения, — каждый объект рано или поздно достигнет точки, в которой силовое поле достаточно велико для того, чтобы замедлить его и вернуть в исходную точку. Подтвердим это умозаключение, реализовав гравитационное поле черной дыры в игре с астероидами.

11.3. ДОБАВЛЕНИЕ ГРАВИТАЦИИ В ИГРУ С АСТЕРОИДАМИ

Черная дыра в нашей игре — это объект `PolygonModel` с 20 вершинами, равноудаленными от центра, поэтому он будет выглядеть почти круглым. Сила гравитационного притяжения черной дыры будет определяться одним числом, которое мы назовем гравитацией. Это число передается конструктору черной дыры:

```
class BlackHole(PolygonModel):
    def __init__(self,gravity):
        vs = [vectors.to_cartesian((0.5, 2 * pi * i / 20))
              for i in range(0,20)]
        super().__init__(vs)
        self.gravity = gravity #<2>
```

Определяет вершины BlackHole как PolygonModel

Обратите внимание на то, что все 20 вершин черной дыры находятся на расстоянии 0,5 единицы от начала координат под одинаковыми углами, поэтому она выглядит почти круглой. Добавив следующую строку:

```
black_hole = BlackHole(0.1)
```

мы создадим объект `BlackHole` с центром в начале координат и значением гравитации (`gravity`) 0,1. Чтобы черная дыра появилась на экране (рис. 11.12), ее нужно рисовать в каждой итерации игрового цикла. Далее я добавлю в функцию `draw_poly` именованный аргумент `fill`, чтобы обеспечить заливку черной дыры и сделать ее черной:

```
draw_poly(screen, black_hole, fill=True)
```

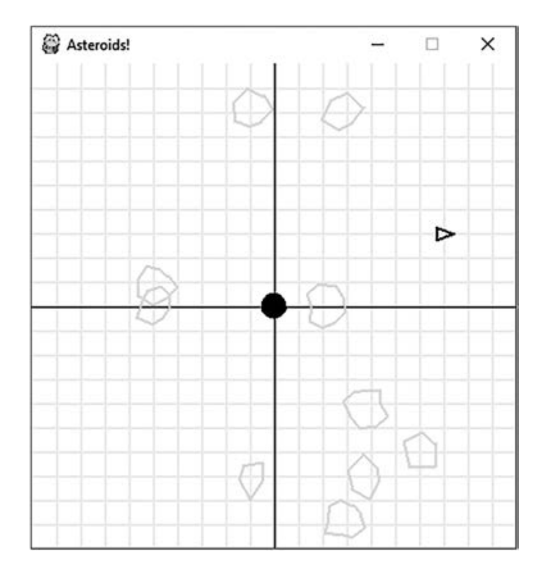


Рис. 11.12. Создание черной дыры в центре игрового поля

Гравитационное поле, создаваемое черной дырой, определяется векторным полем $\mathbf{F}(x, y) = (-x, -y)$, которое во всех точках указывает на начало координат. Если центр черной дыры находится в точке (x_{bh}, y_{bh}), то векторное поле $\mathbf{g}(x, y) = (x_{bh} - x, y_{bh} - y)$ во всех точках будет направлено от (x, y) к (x_{bh}, y_{bh}). Это означает, что стрелка, прикрепленная к точке (x, y), будет указывает на центр черной дыры. Чтобы сила воздействия поля масштабировалась в соответствии с гравитационной силой черной дыры, умножим векторы векторного поля на значение гравитации:

```
def gravitational_field(source, x, y):
    relative_position = (x - source.x, y - source.y)
    return vectors.scale(-source.gravity, relative_position)
```

В этой функции `source` — это объект `BlackHole`, его свойства `x` и `y` задают центр `PolygonModel`, а свойство `gravity` — это значение, переданное в конструкторе. Эквивалентная математическая запись силового поля выглядит так:

$$\mathbf{g}(x, y) = G_{\text{bh}} \cdot (x - x_{\text{bh}}, y - y_{\text{bh}}).$$

Здесь G_{bh} представляет выдуманную гравитационную характеристику черной дыры, а $(x_{\text{bh}}, y_{\text{bh}})$ — ее местоположение. Следующий шаг — использовать это гравитационное поле для определения параметров движения объектов.

11.3.1. Реализация воздействия гравитации на игровые объекты

Если векторное поле работает как гравитационное, оно сообщает нам силу, действующую на единицу массы объекта в точке (x, y). Иначе говоря, на объект с массой m будет действовать сила $\mathbf{F}(x, y) = m \cdot \mathbf{g}(x, y)$. Если это единственная сила, действующая на объект, то мы можем вычислить его ускорение, используя второй закон Ньютона:

$$\mathbf{a} = \frac{\mathbf{F}_{\text{net}}(x, y)}{m} = \frac{m \cdot \mathbf{g}(x, y)}{m}.$$

В последнем выражении, определяющем ускорение, масса тела m присутствует и в числителе, и в знаменателе, поэтому она сокращается. Получается, что вектор гравитационного поля равен вектору ускорения, вызванного гравитацией, и не зависит от массы объекта. Эта формула работает и для реальных гравитационных полей, именно поэтому все объекты разной массы падают с одинаковой скоростью около 9,81 м/с вблизи поверхности Земли. В одной итерации игрового цикла с приращением времени Δt изменение скорости космического корабля или астероида определяется его положением (x, y) как

$$\Delta\mathbf{v} = \mathbf{a} \cdot \Delta t = \mathbf{g}(x, y) \cdot \Delta t.$$

Теперь добавим код, корректирующий скорость космического корабля, а также каждого астероида в каждой итерации игрового цикла. Организовать этот код можно несколькими способами, но я предпочитаю инкапсулировать всю физику в метод `move` объектов `PolygonModel`. Возможно, вы также помните, что мы телепортировали объекты на противоположную сторону игрового поля, чтобы они не улетали за пределы экрана. Я добавлю еще одно небольшое изменение — глобальный флаг `bounce`, который определяет поведение объектов по достижении границы игрового поля — телепортация или отскок. Я сделал это потому, что сразу после телепортации объекты начинают испытывать иное гравитационное воздействие, если же они будут отскакивать от краев, то мы получим более понятную физику. Вот обновленный метод `move`:

```
def move(self, milliseconds,
         thrust_vector, gravity_source):
    tx, ty = thrust_vector
    gx, gy = gravitational_field(src, self.x, self.y)
    ax = tx + gx
    ay = ty + gy
    self.vx += ax * milliseconds/1000
    self.vy += ay * milliseconds/1000

    self.x += self.vx * milliseconds / 1000.0
    self.y += self.vy * milliseconds / 1000.0

    if bounce:
        if self.x < -10 or self.x > 10:
            self.vx = - self.vx
        if self.y < -10 or self.y > 10:
            self.vy = - self.vy
    else:
        if self.x < -10:
            self.x += 20
        if self.y < -10:
            self.y += 20
        if self.x > 10:
            self.x -= 20
        if self.y > 10:
            self.y -=20
```

thrust_vector — это вектор тяги, который может иметь значение (0,0), а gravity_source — источник гравитации (черная дыра)

Здесь результирующая сила вычисляется как сумма векторов тяги и гравитационной силы. Предполагается, что масса равна 1, а ускорение равно сумме тяги и гравитационного поля

Скорость корректируется, как и прежде, с использованием Δv = a*Δt

Обновляем положение вектора, как и ранее, используя Δs = v*Δt

Если глобальный флаг bounce имеет значение True, то нужно изменить знак компоненты скорости x, когда объект собирается покинуть игровое поле слева или справа, или компоненты скорости y, когда объект собирается покинуть игровое поле вверху или внизу

Иначе использовать тот же эффект телепортации, что и раньше, если объект собирается покинуть игровое поле

Осталось только вызвать этот метод для космического корабля, а также для каждого астероида в игровом цикле:

```
while not done:
    ...
    for ast in asteroids:
        ast.move(milliseconds, (0,0), black_hole)

    thrust_vector = (0,0)
    if keys[pygame.K_UP]:
        thrust_vector=vectors.to_cartesian((thrust, ship.rotation_angle))

    elif keys[pygame.K_DOWN]:
        thrust_vector=vectors.to_cartesian((-thrust, ship.rotation_angle))

    ship.move(milliseconds, thrust_vector, black_hole)
```

Для каждого астероида вызывается свой метод move с вектором тяги, равным (0,0)

Вектор тяги корабля по умолчанию также равен (0,0)

Вызвать метод move космического корабля, чтобы заставить его двигаться

Если нажата клавиша со стрелкой вверх или вниз, вычислить вектор тяги, используя направление космического корабля и фиксированное скалярное значение тяги

Запустив игру сейчас, вы увидите, что объекты начинают притягиваться к черной дыре, в том числе и космический корабль, сначала неподвижный, начинает падать прямо в нее! На рис. 11.13 показана пошаговая съемка ускорения корабля.

При любой другой начальной скорости и отсутствии тяги корабль начинает вращаться вокруг черной дыры по эллиптической орбите (рис. 11.14).

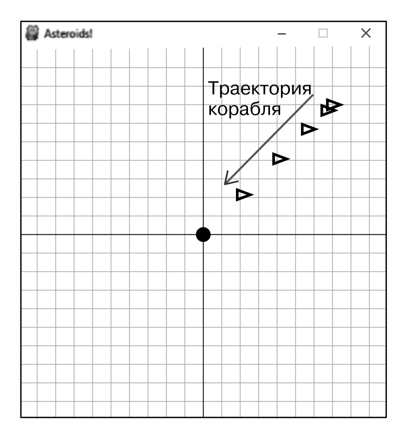


Рис. 11.13. Изначально неподвижный космический корабль начинает падать в черную дыру

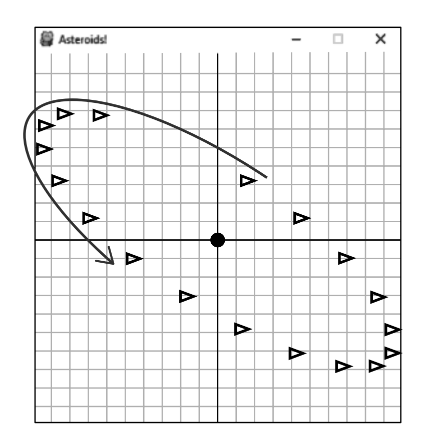


Рис. 11.14. Если корабль имел некоторую начальную скорость, то он начинает вращаться вокруг черной дыры по эллиптической орбите

Получается, что любой объект, не испытывающий никаких сил, кроме гравитации черной дыры, либо падает прямо в нее, либо начинает вращаться по эллиптической орбите. На рис. 11.15 показаны астероид и космический корабль, инициализированные случайными начальными значениями местоположения и скорости. Как видите, они оба движутся по эллиптическим орбитам.

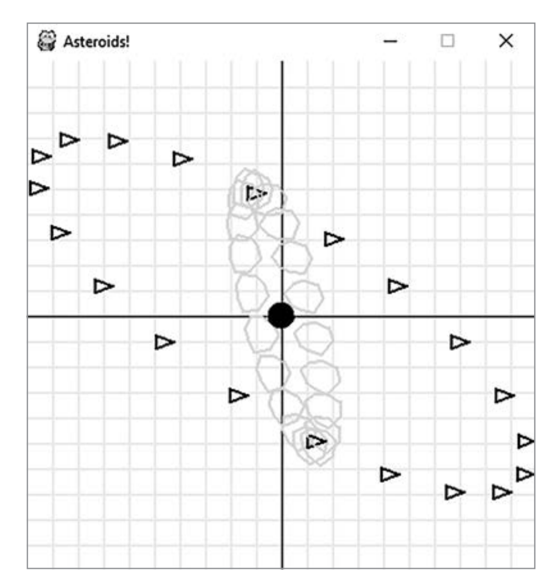


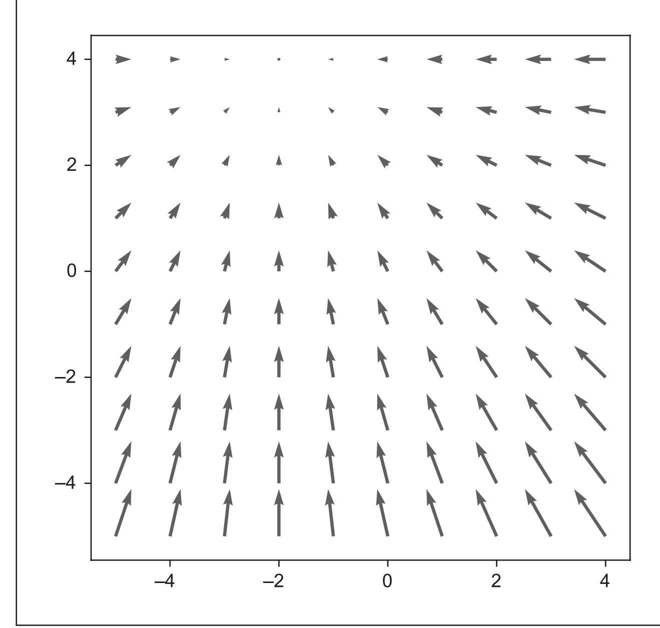
Рис. 11.15. Астероид, инициализированный некоторой начальной скоростью, тоже вращается вокруг черной дыры по эллиптической орбите

Можете попробовать вернуть все астероиды, и вы увидите, что с 11 одновременно ускоряющимися объектами игра стала намного интереснее!

11.3.2. Упражнения

Упражнение 11.1. Куда указывают все векторы векторного поля $(-2 - x, 4 - y)$? Постройте это векторное поле, чтобы подтвердить ответ.

Решение. Это векторное поле совпадает с вектором смещения $(-2, 4) - (x, y)$, который указывает из точки (x, y) в точку $(-2, 4)$. Поэтому ожидается, что каждый вектор в этом векторном поле будет указывать в точку $(-2, 4)$. Отображение векторного поля подтверждает это.



Упражнение 11.2. Мини-проект. Пусть есть *две* черные дыры, обе с силой гравитации 0,1. Они расположены в точках $(-3, 4)$ и $(2, 1)$. Соответственно, гравитационные поля будут выражаться как $\mathbf{g}_1(x, y) = 0{,}1 \cdot (-3 - x, 4 - y)$ и $\mathbf{g}_2(x, y) = 0{,}1 \cdot (2 - x, 1 - y)$. Вычислите формулу полного гравитационного поля $\mathbf{g}(x, y)$, создаваемого обеими черными дырами. Эквивалентно ли оно полю, создаваемому одной черной дырой? Если да, то почему?

Решение. В каждой точке (x, y) на объект с массой m действуют две гравитационные силы, $m \cdot \mathbf{g}_1(x, y)$ и $m \cdot \mathbf{g}_2(x, y)$. Векторная сумма этих сил равна $m(\mathbf{g}_1(x, y) + \mathbf{g}_2(x, y))$. На единицу массы будет действовать сила, равная $\mathbf{g}_1(x, y) + \mathbf{g}_2(x, y)$, и это подтверждает, что вектор полного гравитационного поля является суммой векторов гравитационных полей каждой из черных дыр. Полное гравитационное поле составляет

$$\mathbf{g}(x, y) = \mathbf{g}_1(x, y) + \mathbf{g}_2(x, y) =$$
$$= 0{,}1 \cdot (-3 - x, 4 - y) + 0{,}1 \cdot (2 - x, 1 - y).$$

Упростим и вынесем общий множитель 2 за скобки:

$$\mathbf{g}(x, y) = 0{,}1 \cdot 2 \cdot (-0{,}5 - x, 2{,}5 - y) =$$
$$= 0{,}2 \cdot (-0{,}5 - x, 2{,}5 - y).$$

Это поле соответствует одиночной черной дыре с силой гравитации 0,2, расположенной в точке (–0,5, 2,5).

Упражнение 11.3. Мини-проект. Добавьте в игру с астероидами две черные дыры и реализуйте влияние их сил гравитации друг на друга. Затем реализуйте их гравитационное влияние на астероиды и космический корабль.

Решение. Полную реализацию вы найдете в примерах исходного кода. Ключевым дополнением является вызов метода `move` каждой черной дыры в каждой итерации игрового цикла с передачей списка всех остальных черных дыр как источников гравитации:

```
for bh in black_holes:
    others = [other for other in black_holes if other != bh]
    bh.move(milliseconds, (0,0), others)
```

11.4. ПОТЕНЦИАЛЬНАЯ ЭНЕРГИЯ

Теперь, получив представление о том, как ведут себя космический корабль и астероиды в нашем гравитационном поле, мы можем построить вторую модель — их поведения с учетом потенциальной энергии. В игре с астероидами уже есть черные дыры, поэтому в оставшейся части главы мы займемся исследованием

математической основы. Векторные поля, в том числе гравитационные, часто определяются вычислительной операцией, называемой градиентом. Она станет для нас одним из основных инструментов в следующих главах книги.

Суть заключается в следующем: вместо изображения гравитации в виде вектора силы в каждой точке, притягивающей объекты к источнику, объекты в гравитационном поле можно рассматривать как шарики, перекатывающиеся в чаше. Шарики могут катиться в любом направлении, но их всегда тянет на дно чаши. Форму такой воображаемой чаши определяет функция потенциальной энергии. Как выглядит чаша, можно увидеть в центре на рис. 11.16.

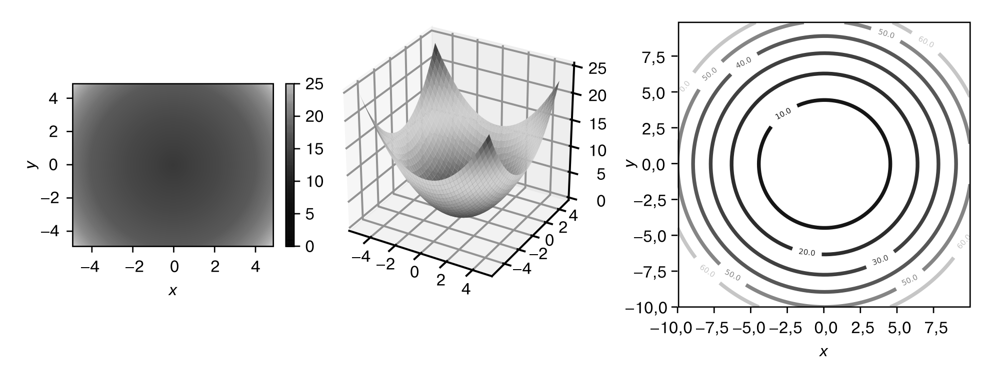


Рис. 11.16. Три изображения скалярного поля: тепловая карта, график и карта рельефа

Запишем потенциальную энергию как функцию, принимающую точку (x, y) и возвращающую единственное число, представляющее гравитационную потенциальную энергию в этой точке. В аналогии с чашей это что-то вроде высоты стенки чаши в данной точке. Реализовав функцию потенциальной энергии на Python, мы сможем визуализировать ее тремя способами: в виде тепловой карты, как было показано в начале главы, в виде трехмерного графика и в виде карты рельефа, как показано на рис. 11.16.

Эти способы визуализации помогут наглядно изобразить функции потенциальной энергии в последнем разделе данной главы и в остальных главах книги.

11.4.1. Определение скалярного поля потенциальной энергии

Скалярное поле, подобно векторному, можно представить в виде функции, принимающей точки (x, y). Однако вместо векторов эта функция возвращает скаляры. Поэкспериментируем с функцией $U(x, y) = (1/2)(x^2 + y^2)$,

определяющей скалярное поле. На рис. 11.17 показано, что на вход этой функции можно передать двухмерный вектор и получить на выходе некоторый скаляр, определяемый формулой $U(x, y)$.

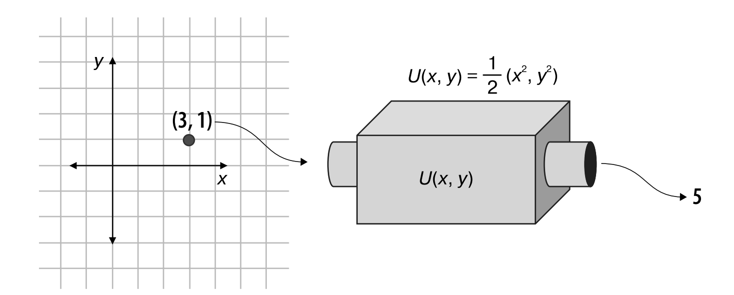


Рис. 11.17. Скалярное поле как функция принимает точку на плоскости и выдает соответствующее число. В данном случае для $(x, y) = (3, 1)$ значением $U(x, y)$ является $(1/2) \cdot (3^2 + 1^2) = 5$

Функция $U(x, y)$ — это функция потенциальной энергии, соответствующая векторному полю $\mathbf{F}(x, y) = (-x, -y)$. Мне пришлось бы немало потрудиться, чтобы объяснить это на языке математики, поэтому попробую показать это наглядно, на качественном уровне, изобразив скалярное поле $U(x, y)$.

Один из способов изобразить $U(x,y)$ — нарисовать трехмерный график (рис. 11.18), где $U(x, y)$ — это поверхность из точек (x, y, z), а $z = U(x, y)$. Например, $U(3, 1) = 5$, поэтому нанесем над точкой (3, 1) в плоскости xy первую точку с координатой $z = 5$.

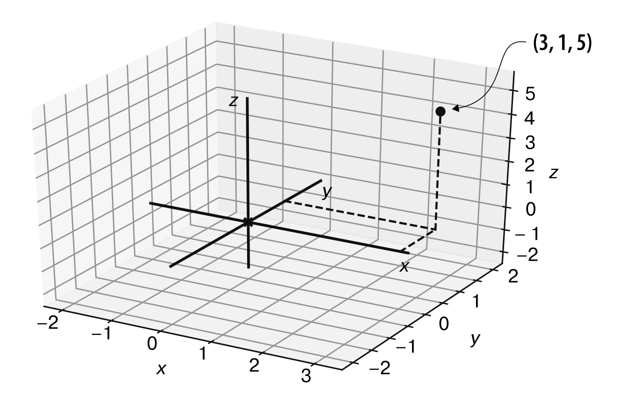


Рис. 11.18. Чтобы нарисовать одну точку $U(x, y) = (1/2)(x^2 + y^2)$, возьмем точку $(x, y) = (3, 1)$ и используем $U(3, 1) = 5$ в качестве координаты z

Нарисовав в трехмерном пространстве точки для каждого отдельного значения (x, y), получим всю поверхность, представляющую скалярное поле $U(x, y)$ и показывающую, как оно изменяется на плоскости. В примерах исходного кода вы найдете функцию `plot_scalar_field`, которая принимает функцию, определяющую скалярное поле, а также границы x и y, и рисует трехмерную поверхность из точек, представляющих поле:

```
def u(x,y):
    return 0.5 * (x**2 + y**2)

plot_scalar_field(u, -5, 5, -5, 5)
```

Существует несколько способов визуализации скалярного поля, однако я буду ссылаться на график функции $U(x, y)$, изображенный на рис. 11.19.

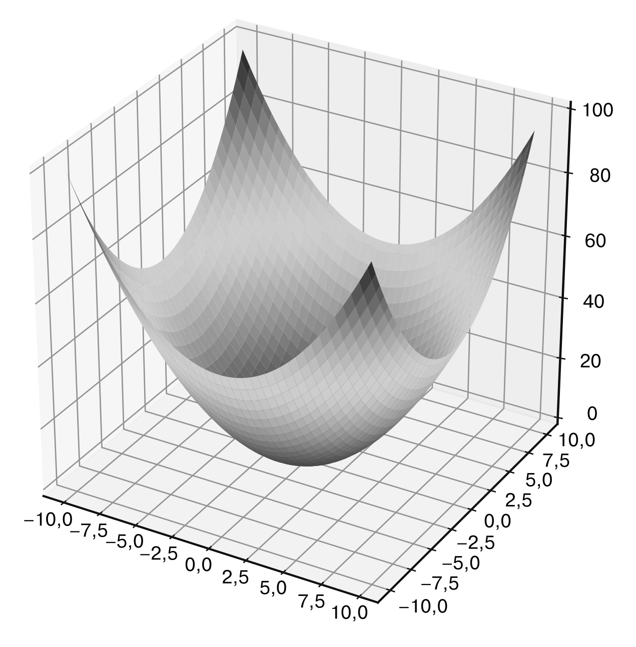


Рис. 11.19. График скалярного поля потенциальной энергии $U(x, y) = (1/2)(x^2 + y^2)$

Это та же самая чаша из предыдущей аналогии. Оказывается, эта функция потенциальной энергии дает ту же модель гравитации, что и векторное поле $\mathbf{F}(x, y) = (-x, -y)$. Почему это именно так, вы увидите в разделе 11.5, а сейчас мы просто подтвердим, что потенциальная энергия увеличивается с увеличением расстояния от начала координат $(0, 0)$. Во всех радиальных направлениях высота графика увеличивается, что означает увеличение значения U.

11.4.2. Представление скалярного поля в виде тепловой карты

Еще один способ визуализировать скалярную функцию — нарисовать тепловую карту. Вместо координаты z для визуализации значения $U(x, y)$ можно использовать цветовую схему. Это позволит изобразить скалярное поле в двухмерном пространстве. Благодаря наличию цветовой шкалы сбоку (рис. 11.20) можно приблизительно оценить скалярное значение в точке (x, y) по цвету в ней на графике.

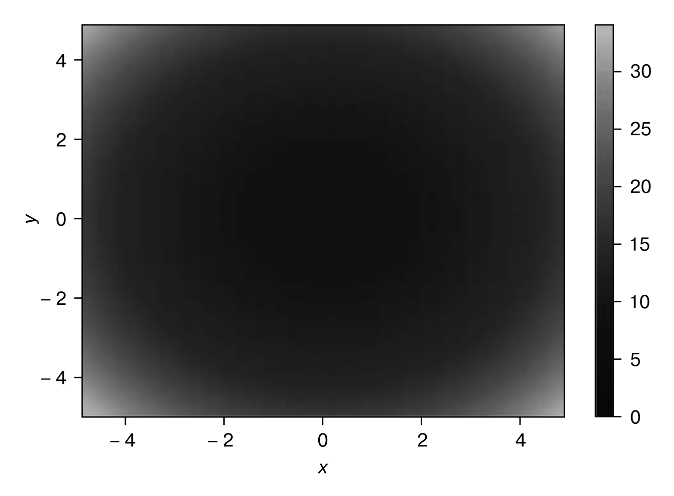


Рис. 11.20. Тепловая карта функции $U(x, y)$

В центре графика, около начала координат $(0, 0)$, цвет темнее, что соответствует более низким значениям $U(x, y)$. На краях цвет светлее, что означает более высокие значения $U(x, y)$. Можете нарисовать тепловую карту функции потенциальной энергии самостоятельно, использовав функцию `scalar_field_heatmap` из примеров с исходным кодом.

11.4.3. Представление скалярного поля в виде карты рельефа

Карта рельефа похожа на тепловую карту. Возможно, вы уже видели подобные карты рельефа — топографические карты, которые показывают высоту местности над уровнем моря. Карта этого типа состоит из изолиний, соединяющих точки с одинаковой высотой, поэтому, если идти вдоль изолинии, нанесенной на карту, вы не будете смещаться ни вверх, ни вниз. На рис. 11.21 показана карта рельефа для $U(x, y)$, где на плоскости xy изображены изолинии, соединяющие точки с одинаковыми значениями $U(x, y)$, равными 10, 20, 30, 40, 50 и 60.

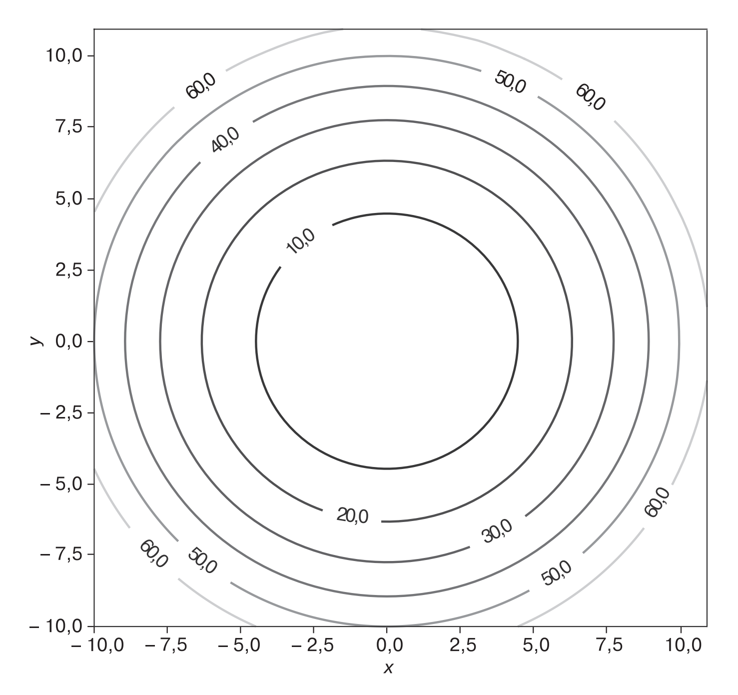


Рис. 11.21. Карта рельефа для функции *U*(*x*, *y*) с изолиниями, соединяющими точки с одинаковыми значениями *U*(*x*, *y*)

Видно, что все изолинии круглые и становятся тем ближе друг к другу, чем ближе находятся к краю. Это обстоятельство можно интерпретировать как то, что $U(x, y)$ становится круче по мере удаления от начала координат. Например, расстояние между изолиниями 30 и 40 меньше, чем между изолиниями 10 и 20. Построить скалярное поле U в виде карты рельефа можно с помощью функции `scalar_field_contour` из примеров с исходным кодом.

11.5. СВЯЗЬ ЭНЕРГИИ И СИЛ С ГРАДИЕНТОМ

Существует важное понятие *крутизны* — крутизна функции потенциальной энергии говорит о том, сколько энергии должен приложить объект, чтобы продвинуться в заданном направлении. Как и следовало ожидать, усилие, необходимое для движения в некотором направлении, является мерой силы, действующей *в противоположном направлении*. В оставшейся части этого раздела мы рассмотрим точную и количественную версии этого утверждения.

Как упоминалось во введении к главе, градиент — это операция, которая принимает скалярное поле, подобное полю потенциальной энергии, и создает векторное поле, подобное полю гравитации. В каждой точке (x, y) на плоскости векторное поле градиента указывает направление наибольшего увеличения скалярного

поля. В этом разделе я покажу, как взять градиент скалярного поля $U(x, y)$, для чего необходимо получить производную U отдельно по x и по y. Так мы сможем показать, что градиент функции потенциальной энергии $U(x, y)$ из нашего примера равен $-\mathbf{F}(x, y)$, где $\mathbf{F}(x, y)$ — это гравитационное поле, реализованное в игре с астероидами. Мы будем широко использовать градиент в оставшихся главах.

11.5.1. Измерение крутизны с помощью поперечных сечений

Есть еще один способ визуализации функции $U(x, y)$, позволяющий оценить ее крутизну в различных точках. Возьмем для примера конкретную точку $(x, y) = (-5, 2)$. На карте рельефа, подобной показанной на рис. 11.22, она находится между изолиниями $U = 10$ и $U = 20$, и фактически $U(-5, 2) = 14{,}5$. Если двигаться в направлении положительного конца оси x, мы попадаем на изолинию $U = 10$, а это означает, что U уменьшается в направлении $+x$. Если двигаться в направлении $+y$, мы попадем на изолинию $U = 20$, то есть U увеличивается в этом направлении.

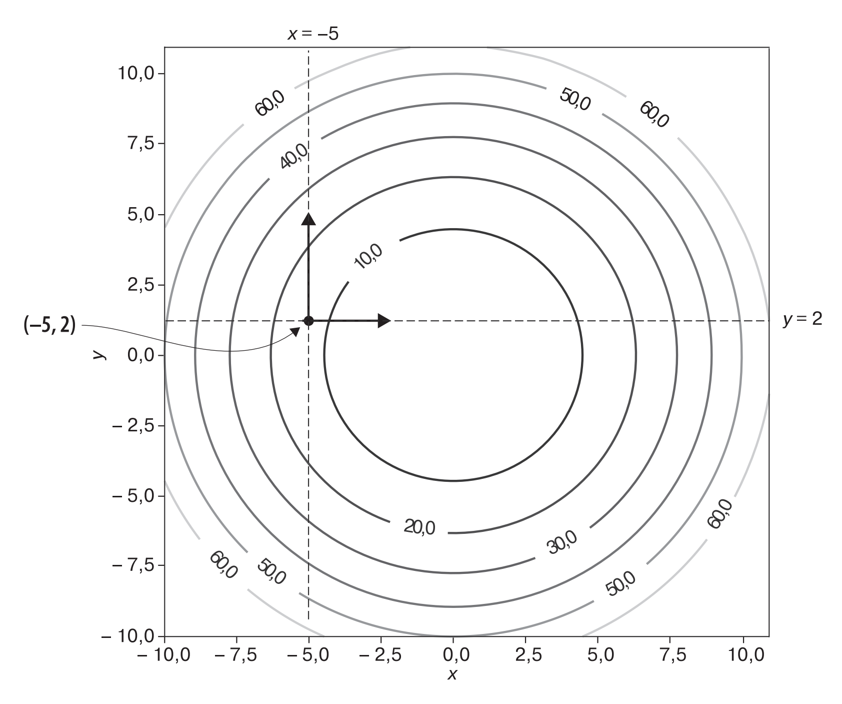


Рис. 11.22. Как изменяется значение $U(x, y)$ при движении в направлениях $+x$ и $+y$ из точки (–5, 2)

На рис. 11.22 показано, что крутизна $U(x, y)$ зависит от направления. Это можно изобразить, построив сечения $U(x, y)$ при $x = -5$ и $y = 2$. *Сечения* — это срезы графика $U(x, y)$ при фиксированных значениях x или y. Например, на рис. 11.23

показано поперечное сечение $U(x, y)$ при $x = -5$, которое является срезом $U(x, y)$ в плоскости $x = -5$.

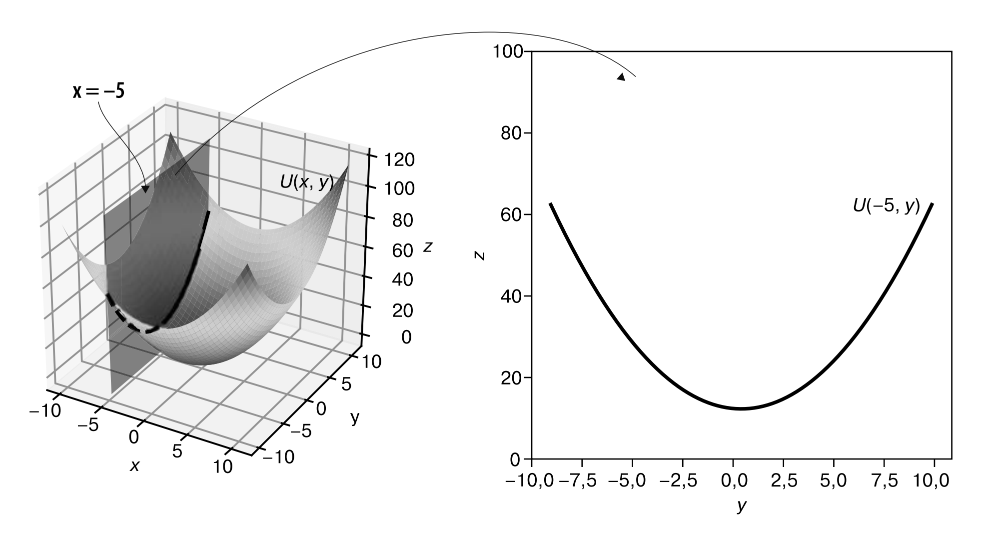


Рис. 11.23. Сечение $U(x, y)$ при $x = -5$

Используя терминологию функционального программирования из главы 4, мы можем частично применить U с $x = -5$, чтобы получить функцию, которая принимает одно число y и возвращает значение U. Аналогично можно нарисовать поперечное сечение параллельно оси y, проходящее через точку $(5, 2)$. Это поперечное сечение $U(x, y)$ при $y = 2$. На рис. 11.24 показана его форма в виде графика $U(x, y)$ после частичного применения с $y = 2$.

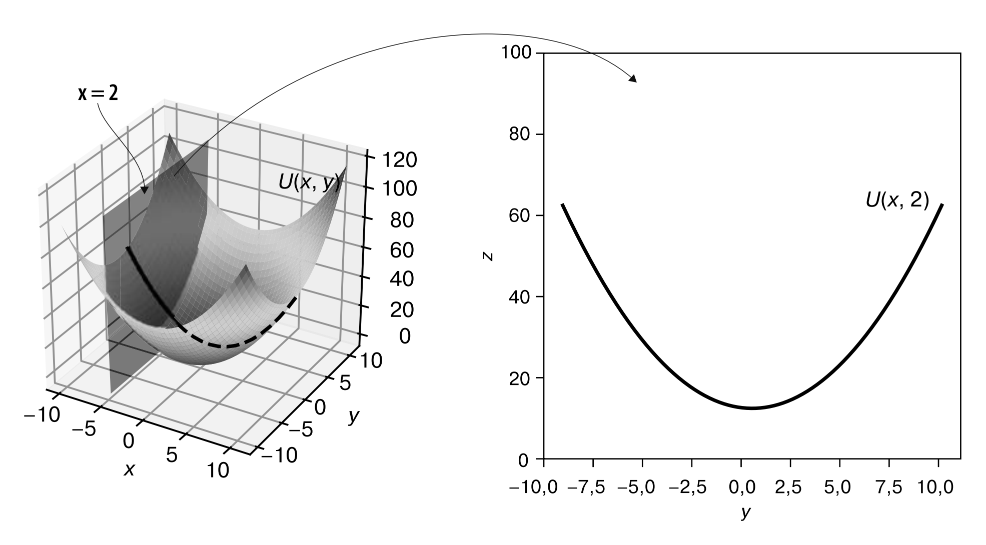


Рис. 11.24. Сечение $U(x, y)$ при $y = 2$

Вместе эти сечения говорят о том, как U изменяется в точке $(-5, 2)$ в направлениях x и y. Наклон $U(x, 2)$ в точке $x = -5$ отрицателен, то есть при движении в направлении $+x$ из точки $(-5, 2)$ значение U будет уменьшаться. В то же время наклон $U(-5, y)$ в точке $y = 2$ положителен, то есть при движении в направлении $+y$ из точки $(-5, 2)$ значение U будет увеличиваться (рис. 11.25).

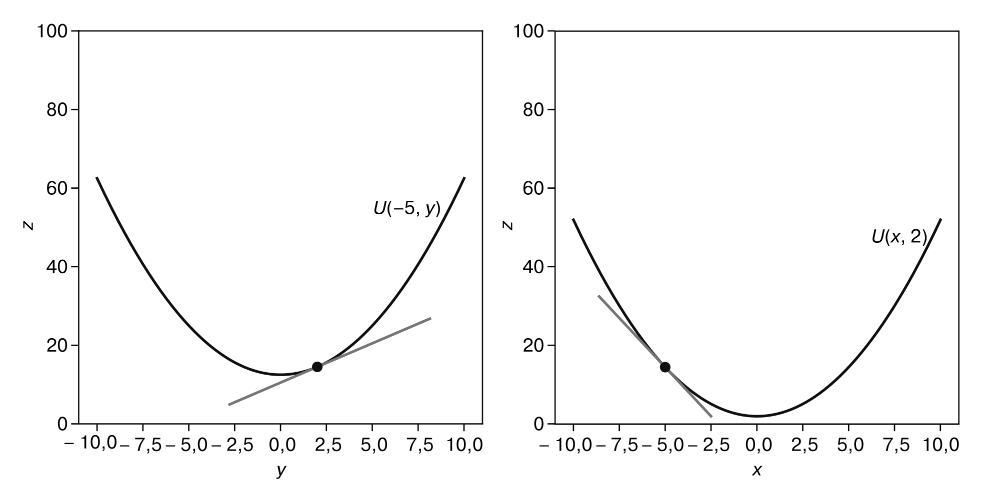


Рис. 11.25. Поперечные сечения показывают, что $U(x, y)$ увеличивается в направлении $+y$ и уменьшается в направлении $+x$

Мы пока не нашли наклон скалярного поля $U(x, y)$ в этой точке, но нашли то, что можно назвать наклоном в направлении x и наклоном в направлении y. Эти значения называются *частными производными* U.

11.5.2. Расчет частных производных

Вы уже знаете все, что нужно знать, чтобы найти предыдущие наклоны. Обе функции, $U(-5, y)$ и $U(x, 2)$, являются функциями одной переменной, поэтому мы можем аппроксимировать их производные, вычислив наклон небольших секущих.

Например, чтобы найти частную производную $U(x, y)$ по x в точке $(-5, 2)$, нужно найти наклон $U(x, 2)$ в точке $x = -5$. То есть мы должны узнать, насколько быстро $U(x, y)$ изменяется в направлении x в точке $(x, y) = (-5, 2)$. Можно было бы аппроксимировать эту скорость, вставив небольшое значение Δx в следующую формулу вычисления наклона:

$$\frac{U(-5+\Delta x, 2)-U(-5, 2)}{\Delta x}.$$

Можно также точно вычислить производную, записав формулу для $U(x, 2)$. Исходя из $U(x, y) = (1/2)(x^2 + y^2)$, получаем $U(x, 2) = (1/2)(x^2 + 2^2) = (1/2)(x^2 + 4) = 2 + (x^2/2)$. Используя степенное правило вычисления производных, получаем производную для $U(x, 2)$ по x: $0 + 2x/2 = x$. При $x = -5$ производная равна -5.

Обратите внимание на то, что ни в аппроксимации наклона, ни в символьном вычислении производной нет переменной y. Вместо нее берется константа 2. Этого и следовало ожидать, потому что в частной производной в направлении x значение y не меняется. Обобщенный способ символьного вычисления частных производных состоит в том, чтобы взять производную так, будто только один символ (например, x) — это переменная, а все остальные (например, y) — константы.

Согласно этому методу частная производная $U(x, y)$ по x составляет $(1/2)(2x + 0) = x$, а частная производная по y — $(1/2)(0 + 2y) = y$. Кстати, обозначения $f'(x)$, которое мы использовали ранее для представления производной функции $f(x)$, недостаточно для представления частных производных. Частная производная может браться по разным переменным, поэтому нужно указать, по какой именно. Есть другое эквивалентное обозначение производной $f'(x)$:

$$f'(x) \equiv \frac{df}{dx}.$$

(Я использую знак ≡, чтобы указать, что эти обозначения эквивалентны — они представляют одно и то же понятие.) Эта формула напоминает формулу наклона $\Delta f/\Delta x$, но в данном обозначении члены df и dx представляют *бесконечно малые* изменения значений f и x. Обозначение df/dx — суть то же самое, что и $f'(x)$, но оно более четко указывает, что производная берется по x. Для функции, такой как $U(x, y)$, мы можем взять частную производную либо по x, либо по y. Чтобы показать, что берется не обычная производная, которую иногда называют *полной* производной, традиционно применяется буква «∂» другой формы. Частные производные U по x и y соответственно записываются следующим образом:

$$\frac{\partial U}{\partial x} = x; \ \frac{\partial U}{\partial y} = y.$$

Вот еще один пример с функцией $q(x, y) = x \sin(xy) + y$. Если рассматривать y как константу и взять производную по x, то мы должны использовать правило произведения и цепное правило. Результатом является частная производная по x:

$$\frac{\partial q}{\partial x} = \sin(xy) + xy\cos(xy).$$

Чтобы взять частную производную по y, значение x рассматривается как константа, и нужно использовать цепное правило и правило сумм:

$$\frac{\partial q}{\partial y} = x^2 \cos(xy) + 1.$$

Воистину, каждая из частных производных рассказывает только часть истории о том, как изменяется в любой точке такая функция, как $U(x, y)$. Чтобы получить полное представление, подобное полной производной для функции одной переменной, мы должны объединить их.

11.5.3. Определение крутизны графика с использованием градиента

Увеличим окрестности точки $(-5, 2)$ на графике $U(x, y)$ (рис. 11.26). Как известно, на достаточно коротком диапазоне значений x любая гладкая функция $f(x)$ похожа на прямую линию. А в достаточно ограниченной окрестности плоскости xy график гладкого скалярного поля похож на плоскость.

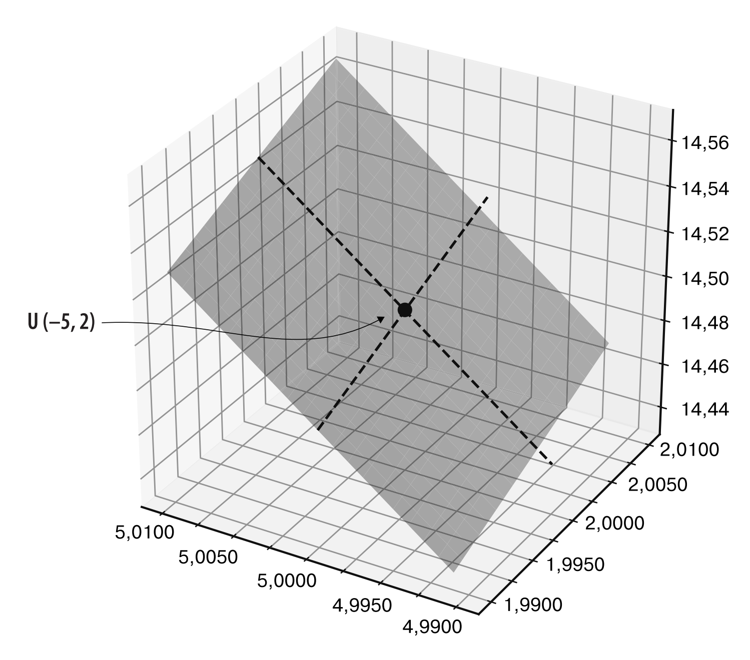


Рис. 11.26. При достаточном увеличении окрестности точки $(x, y) = (-5, 2)$ на графике $U(x, y)$ похожи на плоскость

Точно так же как производная df/dx говорит о наклоне линии, аппроксимирующей $f(x)$ в данной точке, частные производные $\partial U/\partial x$ и $\partial U/\partial y$ говорят об

ориентации плоскости, аппроксимирующей $U(x, y)$ в данной точке. Пунктирные линии на рис. 11.26 показывают сечения $U(x, y)$ по осям x и y в заданной точке. В этом окне они выглядят почти как прямые линии, а их наклоны в плоскостях xz и yz близки к частным производным $\partial U/\partial x$ и $\partial U/\partial y$.

Я не приводил никаких доказательств, но предположим, что существует плоскость, которая лучше всего аппроксимирует $U(x, y)$ вблизи точки $(-5, 2)$, и поскольку различия практически не видны, мы можем представить на мгновение, что график на рис. 11.26 и есть эта плоскость. Частные производные говорят нам, насколько он наклонен в направлениях x и y. На плоскости на самом деле есть два основных направления, о которых следовало бы порассуждать. Во-первых, есть линия, идя вдоль которой мы не будем подниматься или опускаться. Иначе говоря, это линия параллельна плоскости xy. Для плоскости, аппроксимирующей $U(x, y)$ в точке $(-5, 2)$, эта линия параллельна вектору $(2, 5)$, как показано на рис. 11.27.

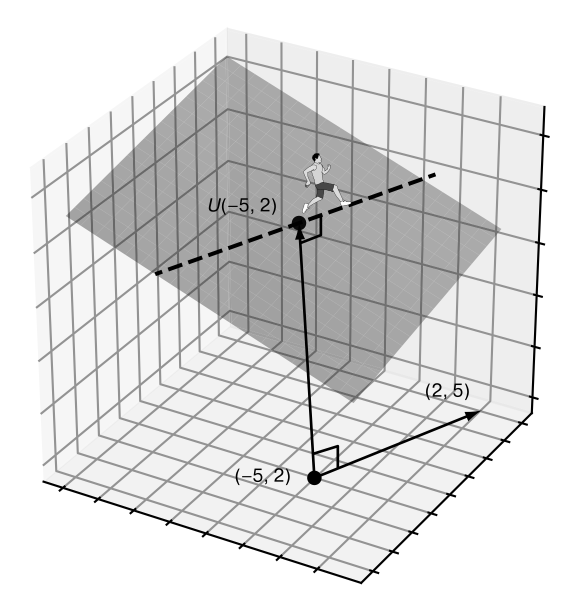


Рис. 11.27. Идя по графику $U(x, y)$ от точки $(x, y) = (-5, 2)$ в направлении $(2, 5)$, мы не наберем и не потеряем высоту

Пешеход, изображенный на рис. 11.27, не особенно затрудняет себя, потому что, двигаясь в этом направлении, не поднимается и не спускается. Однако, если пешеход повернет на 90° влево, он будет идти в гору в самом крутом из возможных направлений. Это направление $(-5, 2)$, перпендикулярное направлению $(2, 5)$.

Как оказывается, направление скорейшего подъема — это вектор, компонентами которого являются частные производные для U в данной точке. Я привел

простую иллюстрацию вместо доказательства, но в целом это наблюдение верно. Вектор частных производных для функции $U(x, y)$ называется *градиентом* и обозначается ∇U. Он задает величину и направление наискорейшего подъема U в данной точке:

$$\nabla U(x, y) = \left(\frac{\partial U}{\partial x}, \frac{\partial U}{\partial y}\right).$$

Имея формулы частных производных, мы можем сказать, например, что для нашей функции $\nabla U(x, y) = (x, y)$. Функция ∇U, являющаяся градиентом U, присваивает вектор каждой точке плоскости, соответственно, это самое настоящее векторное поле! График ∇U показывает в каждой точке (x, y), какое направление является восходящим на графике $U(x, y)$, а также его крутизну (рис. 11.28).

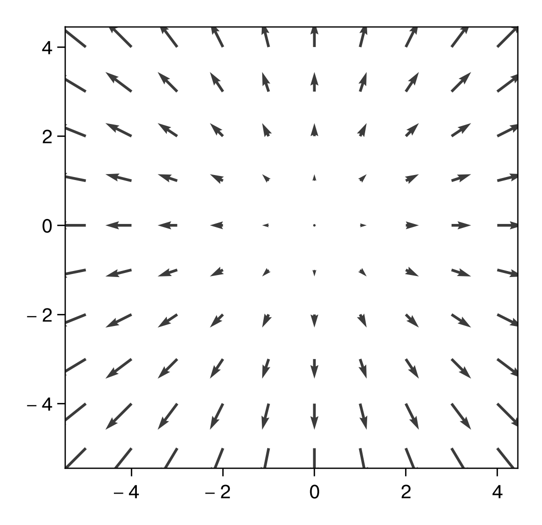


Рис. 11.28. Градиент ∇U — это векторное поле, сообщающее крутизну и направление наискорейшего подъема на графике U в любой точке (x, y)

Градиент помогает связать скалярное и векторное поля. Он отражает связь между потенциальной энергией и силой.

11.5.4. Расчет силовых полей на основе потенциальной энергии с градиентом

Градиент — лучшая аналогия обычной производной для скалярных полей. Он содержит всю информацию, необходимую для определения направления скорейшего подъема скалярного поля, наклона по направлениям x и y или плоскости наилучшей аппроксимации. Но с точки зрения физики направление

скорейшего подъема — это не то, что нам нужно. В конце концов, в природе нет объекта, который самопроизвольно движется в гору.

Ни на космический корабль в игре с астероидами, ни на шарик, катающийся в чаше, не будут воздействовать силы, толкающие их в области с высокой потенциальной энергией. Как мы уже говорили, им придется применить силу или пожертвовать некоторой кинетической энергией, чтобы получить больше потенциальной энергии. По этой причине правильным описанием силы, воздействие которой испытывает объект, является *отрицательный* градиент потенциальной энергии, указывающий в направлении скорейшего спуска, а не подъема. Если $U(x, y)$ представляет собой скалярное поле потенциальной энергии, то соответствующее силовое поле $\mathbf{F}(x, y)$ можно рассчитать по формуле

$$\mathbf{F}(x, y) = -\nabla U(x, y).$$

Разберем другой пример. Какое силовое поле будет создано следующей функцией потенциальной энергии:

$$V(x, y) = 1 + y^2 - 2x^2 + x^6?$$

Чтобы понять, как ведет себя эта функция, построим ее график.

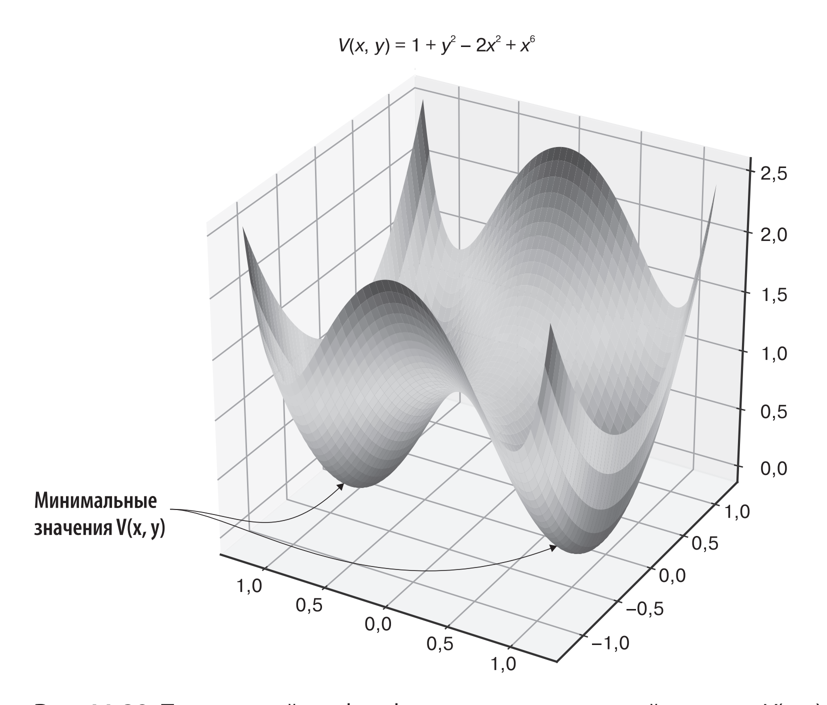


Рис. 11.29. Трехмерный график функции потенциальной энергии $V(x, y)$

Как видно на рис. 11.29, эта функция потенциальной энергии имеет форму двойной чаши с двумя точками минимума и горбом между ними. А как выглядит силовое поле, связанное с этой функцией потенциальной энергии? Чтобы это выяснить, нужно взять отрицательный градиент V:

$$\mathbf{F}(x, y) = -\nabla V(x, y) = -\left(\frac{\partial V}{\partial x}, \frac{\partial V}{\partial y}\right).$$

Мы можем получить частную производную для V по x, рассматривая y как константу, вследствие чего члены 1 и y^2 не вносят вклада. Результат — простая производная от $-2x^2 + x^6$ по x, которая равна $-4x + 6x^5$.

Чтобы получить частную производную для V по y, мы рассматриваем x как константу, поэтому единственный член, который влияет на результат, — это y^2, имеющий производную $2y$. Соответственно, отрицательный градиент $V(x, y)$ составляет

$$\mathbf{F}(x, y) = -\nabla V(x, y) = (4x - 6x^5, -2y).$$

Изображение этого векторного поля на рис. 11.30 показывает, что силовое поле направлено к точкам с наименьшей потенциальной энергией. Объект, на который воздействует это силовое поле, будет воспринимать эти две точки как два источника силы гравитации.

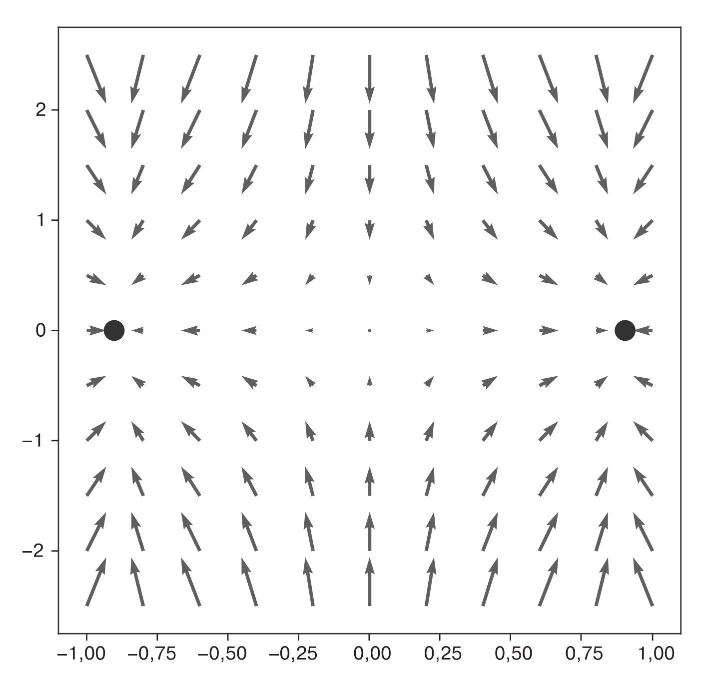


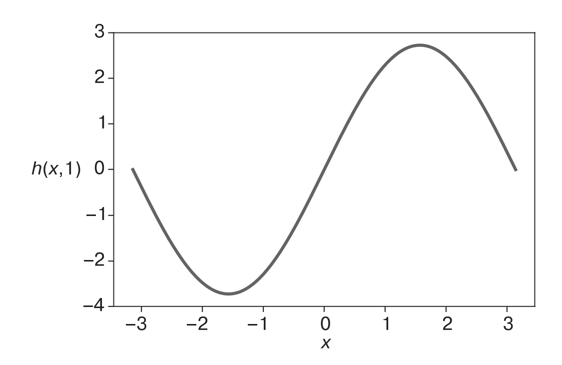
Рис. 11.30. График векторного поля $-\nabla V(x, y)$ представляет силовое поле, связанное с функцией потенциальной энергии $V(x, y)$. Это сила притяжения к двум показанным здесь точкам

Отрицательный градиент потенциальной энергии — это направление, которое предпочитает природа, — направление высвобождения накопленной энергии. Объекты естественным образом стремятся к состояниям, в которых их потенциальная энергия минимальна. Градиент — важный инструмент поиска оптимальных значений скалярных полей, в чем вы убедитесь в следующих главах. В частности, в последней части этой книги я покажу, как следование вдоль отрицательного градиента в поисках оптимального значения имитирует процесс обучения в некоторых алгоритмах машинного обучения.

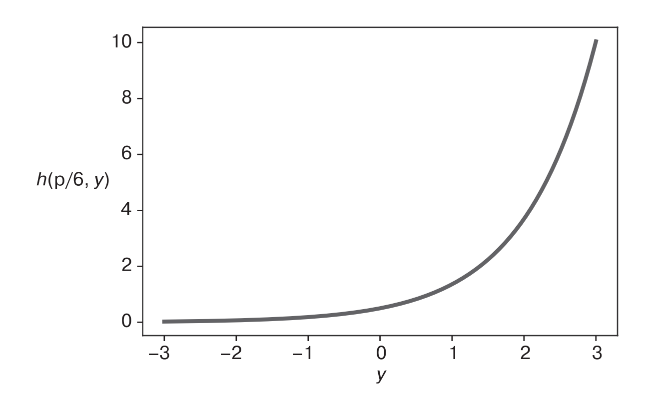
11.5.5. Упражнения

Упражнение 11.4. Постройте сечение $h(x, y) = e^y \sin(x)$ для $y = 1$. Затем постройте сечение $h(x, y)$ для $x = \pi/6$.

Решение. Сечение $h(x, y)$ для $y = 1$ зависит только от x: $h(x, 1) = e^1 \sin(x) = e \cdot \sin(x)$, как показано далее.



При $x = \pi/6$ значение $h(x, y)$ зависит только от y. То есть $h(\pi/6, y) = e^y \sin(\pi/6) = e^y/2$. График сечения выглядит так.



Упражнение 11.5. Приведите частные производные функции $h(x, y)$ из предыдущего упражнения. Что такое градиент? Какое значение имеет градиент при $(x, y) = (\pi/6, 1)$?

Решение. Частная производная для $e^y \sin(x)$ по x получается путем интерпретации y как константы. Соответственно, член e^y рассматривается как константа:

$$\frac{\partial h}{\partial x} = e^y \cos(x).$$

Аналогично, чтобы определить частную производную по y, мы должны интерпретировать x как константу, и, соответственно, $\sin(x)$ становится константой:

$$\frac{\partial h}{\partial y} = e^y \sin(x).$$

Градиент $\nabla h(x, y)$ — это векторное поле, компонентами которого являются частные производные:

$$\nabla h(x, y) = \left(\frac{\partial h}{\partial x}, \frac{\partial h}{\partial y}\right) = \left(e^y \cos(x), e^y \sin(x)\right).$$

При $(x, y) = (\pi/6, 1)$ это векторное поле вычисляется следующим образом:

$$\nabla h\left(\frac{\pi}{6}, 1\right) = \left(e^1 \cos\left(\frac{\pi}{6}\right), e^1 \sin\left(\frac{\pi}{6}\right)\right) = \frac{e}{2} \cdot \left(\sqrt{3}, 1\right).$$

Упражнение 11.6. Докажите, что направление $(-5, 2)$ перпендикулярно направлению $(2, 5)$.

Решение. Этот вопрос обсуждался в главе 2. Эти два вектора перпендикулярны, потому что их скалярное произведение равно нулю: $(-5, 2) \cdot (2, 5) = -10 + 10 = 0$.

Упражнение 11.7. Мини-проект. Пусть $z = p(x, y)$ — уравнение плоскости, наилучшим образом аппроксимирующей $U(x, y)$ в точке $(-5, 2)$. Придумайте (с нуля!) уравнение для $p(x, y)$ и прямой на p, проходящей через точку $(-5, 2)$ и параллельной плоскости xy. Эта прямая должна быть параллельна вектору $(2, 5, 0)$, как я утверждал в предыдущем упражнении.

Решение. Напомню, что $U(x, y)$ определяется формулой $(1/2)(x^2 + y^2)$. Значение $U(-5, 2)$ равно 14,5, то есть это точка $(x, y, z) = (-5, 2, 14{,}5)$ находится на графике $U(x, y)$ в трехмерном пространстве.

Прежде чем переходить к формуле плоскости наилучшей аппроксимации $U(x, y)$, посмотрим, как получить прямую наилучшей аппроксимации функции $f(x)$. Прямая, являющаяся наилучшей аппроксимацией функции $f(x)$ в точке x_0, — это прямая, проходящая через точку $(x_0, f(x_0))$ и имеющая наклон $f'(x_0)$. Эти два факта гарантируют, что значение и производная функции $f(x)$ будут соответствовать аппроксимирующей прямой.

Следуя этой модели, найдем плоскость $p(x, y)$, значение и *обе* частные производные которой соответствуют функции в точке $(x, y) = (-5, 2)$. Это означает, что должны выполняться условия $p(-5, 2) = 14{,}5$, $\partial p/\partial x = -5$ и $\partial p/\partial y = 2$. Уравнение плоскости $p(x, y)$ имеет вид $p(x, y) = ax + by + c$ для некоторых чисел a и b (помните почему?). Частные производные будут выражаться так:

$$\frac{\partial p}{\partial x} = a; \ \frac{\partial p}{\partial y} = b.$$

Чтобы они соответствовали функции, формула должна иметь вид $p(x, y) = -5x + 2y + c$, а чтобы удовлетворить условие $p(-5, 2) = 14{,}5$, должно выполняться равенство $c = -14{,}5$. Отсюда следует, что формула плоскости наилучшей аппроксимации — $p(x, y) = -5x + 2y - 14{,}5$.

Теперь найдем прямую на плоскости $p(x, y)$, проходящую через точку $(-5, 2)$ и параллельную плоскости xy. Это набор точек (x, y), таких, что $p(x, y) = p(-5, 2)$, что означает отсутствие изменения высоты между $(-5, 2)$ и (x, y).

Если $p(x, y) = p(-5, 2)$, то $-5x + 2y - 14{,}5 = -5 \cdot (-5) + 2 \cdot 2 - 14{,}5$. Это условие упрощает уравнение прямой $-5x + 2y = 29$. Данная прямая эквивалентна набору векторов $(-5, 2, 14{,}5) + r \cdot (2, 5, 0)$, где r — действительное число, и, соответственно, она действительно параллельна вектору $(2, 5, 0)$.

КРАТКИЕ ИТОГИ ГЛАВЫ

- Векторное поле — это функция, принимающая и возвращающая вектор. В частности, ее можно представить как функцию, присваивающую вектор-стрелку каждой точке пространства.
- Силу гравитации можно смоделировать векторным полем. Значение векторного поля в любой точке пространства сообщает, насколько сильно и в каком направлении притягивается объект к источнику гравитации.
- Для имитации движения объекта в векторном поле необходимо использовать его положение для расчета силы и направления силового поля в том месте, где он находится. В свою очередь, значение силового поля говорит о силе, действующей на объект, а второй закон Ньютона позволяет определить результирующее ускорение.
- *Потенциальная энергия* — это накопленная энергия, которая может вызвать движение. Потенциальная энергия объекта в силовом поле определяется местоположением объекта в этом поле.
- Потенциальную энергию можно смоделировать как скалярное поле, присвоив каждой точке пространства число, представляющее количество потенциальной энергии объекта в этой точке.
- Существует несколько способов представления скалярного поля в двухмерном пространстве: в виде трехмерной поверхности, тепловой карты, карты рельефа или пары графиков поперечных сечений.
- Частные производные скалярного поля дают скорость изменения значения поля по координатам. Например, если $U(x, y)$ — скалярное поле в двухмерном пространстве, то существуют частные производные по x и y.
- Частные производные совпадают с производными поперечных сечений скалярного поля. Частную производную можно вычислить по одной переменной, рассматривая другие переменные как константы.
- Градиент скалярного поля U — это вектор, компонентами которого являются частные производные U по каждой из координат. Градиент указывает направление скорейшего подъема U, то есть направление, в котором U увеличивается быстрее всего.
- Отрицательный градиент функции потенциальной энергии, соответствующей силовому полю, сообщает векторное значение силового поля в этой точке, подталкивающего объекты к областям с более низкой потенциальной энергией.

12

Оптимизация физической системы

В этой главе

- ✓ Создание модели пушечного ядра и визуализация его полета.
- ✓ Поиск максимального и минимального значений функции с помощью производной.
- ✓ Настройка моделирования с помощью параметров.
- ✓ Визуализация пространств входных параметров для моделирования.
- ✓ Применение градиентного восхождения для максимизации функций нескольких переменных.

В нескольких последних главах основное внимание было сосредоточено на моделировании физики игры. Наша игра — простой и забавный пример, однако есть другие, более важные и прибыльные области применения такого моделирования. Перед началом воплощения любого крупного инженерного проекта, такого как отправка ракеты на Марс, строительство моста или бурение нефтяной скважины, важно знать, насколько он будет безопасным и успешным и уложится ли в отведенный бюджет. В любом проекте есть какие-то количественные параметры, которые было бы желательно оптимизировать. Например, можно попробовать минимизировать время полета ракеты, количество или стоимость бетона, укладываемого при строительстве моста, или максимально увеличить количество нефти, отдаваемое скважиной.

Для более близкого знакомства с оптимизацией сосредоточимся на простом примере моделирования снаряда — ядра, выстреливаемого из пушки. Если предположить, что каждый раз ядро вылетает из ствола с одной и той же скоростью, то траектория полета будет определяться углом выстрела (рис. 12.1).

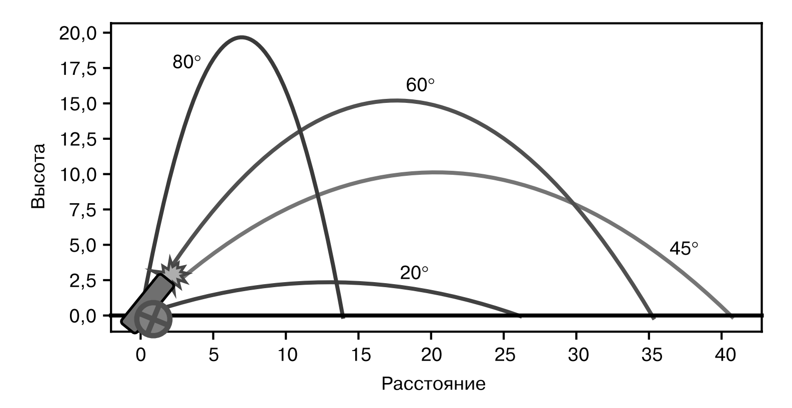


Рис. 12.1. Траектории полета пушечного ядра, выпущенного под разными углами

На рисунке видно, что при выстреле под четырьмя разными углами получаются четыре разные траектории полета. Среди них угол 45° дает наибольшую дальность, а угол 80° — наибольшую высоту полета. Это лишь несколько углов из всех возможных значений от 0 до 90°, поэтому мы не можем утверждать, что они действительно дают наибольшие значения дальности и высоты. Наша задача — систематически исследовать весь диапазон возможных углов запуска и убедиться, что правильно определили тот, который обеспечивает максимальную дальность стрельбы.

Для начала создадим модель пушечного ядра. Реализуем ее в виде функции на Python, принимающей угол выстрела, использующей метод Эйлера (с которым мы познакомились в главе 9) для пошагового моделирования полета пушечного ядра, пока оно не упадет на землю, и возвращающей список его местоположений с течением времени. Из результата извлечем конечную горизонтальную координату ядра, соответствующую координате приземления, то есть дальности полета. Проще говоря, мы реализуем функцию, принимающую угол выстрела и возвращающую дальность полета пушечного ядра под этим углом (рис. 12.2).

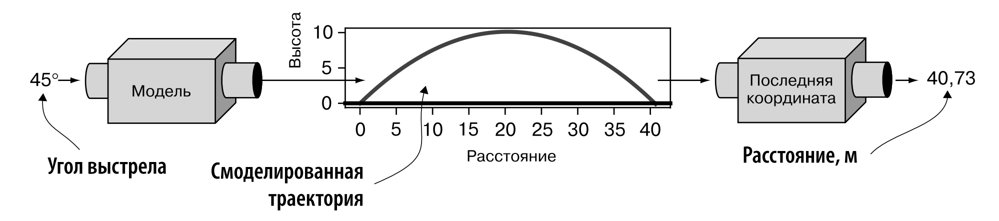


Рис. 12.2. Вычисление дальности полета ядра с помощью приема моделирования

Заключив всю эту логику в функцию на Python `landing_position`, вычисляющую дальность полета пушечного ядра по углу выстрела, мы сможем заняться решением задачи поиска угла, максимизирующего дальность полета. Сделать это можно сделать двумя способами. Первый — построить график зависимости дальности от угла запуска и выбрать наибольшее значение (рис. 12.3).

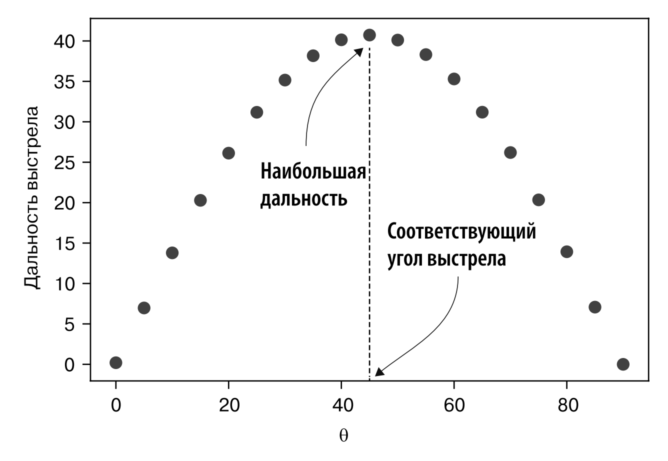


Рис. 12.3. Имея график зависимости дальности от угла выстрела, можно увидеть примерное значение угла, при котором достигается наибольшая дальность

Второй способ поиска оптимального угла выстрела — отставить в сторону нашу модель и найти точную формулу зависимости дальности $r(\theta)$ полета ядра от угла выстрела θ. Она должна дать тот же результат, что и моделирование, но поскольку это математическая формула, для нее можно получить производную, используя правила из главы 10. Производная местоположения приземления по отношению к углу запуска сообщит, насколько увеличится дальность при небольшом увеличении угла. Начиная с некоторого угла дальность начинает уменьшаться — увеличение угла выстрела приводит к *уменьшению* дальности, то есть мы достигаем оптимального значения. Производная $r(\theta)$ становится равной нулю, а значение θ, при котором это происходит, оказывается оптимальным значением.

Попрактиковавшись в применении обоих методов оптимизации в двухмерном пространстве, мы сможем попробовать свои силы в более сложном трехмерном моделировании, где появляется возможность контролировать угол наклона пушки по вертикали, а также угол отклонения вбок. Если стрельба ведется в холмистой местности, то изменение направления может влиять на дальность стрельбы (рис. 12.4).

Для этого примера построим функцию $r(\theta, \phi)$, принимающую два угла, θ и ϕ, и возвращающую координату падения ядра. Цель состоит в том, чтобы найти пару (θ, ϕ), которая увеличивает до максимума дальность выстрела. Этот пример позволит нам рассмотреть третий, самый важный метод оптимизации — *градиентное восхождение*.

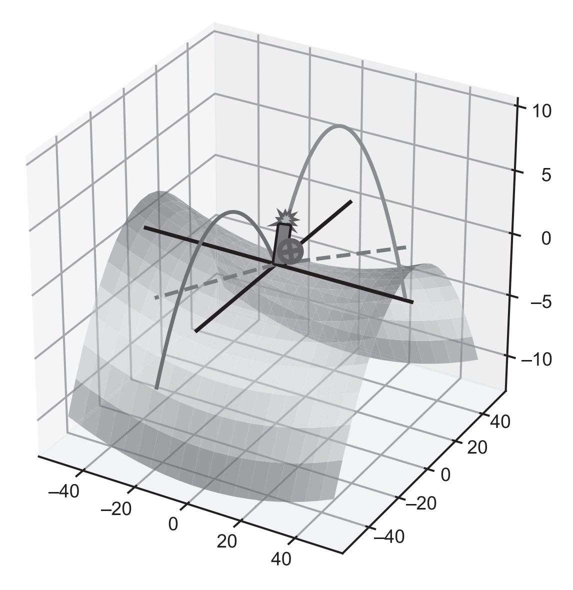


Рис. 12.4. Дальность стрельбы в холмистой местности может также зависеть от направления стрельбы

Как мы узнали в предыдущей главе, градиент $r(\theta, \phi)$ в точке (θ, ϕ) — это вектор, указывающий в направлении скорейшего увеличения r. Напишем на Python функцию `gradient_ascent`, принимающую функцию для оптимизации с парой начальных входных данных для нее и использующую градиент для поиска все более и более высоких значений до достижения оптимальной точки.

Раздел математики, посвященный оптимизации, чрезвычайно обширен, и я надеюсь, что смогу дать вам представление о некоторых основных методах. Все функции, с которыми мы будем работать, являются гладкими, поэтому вы сможете применить к ним все инструменты математического анализа, которые уже изучили. Кроме того, подход к оптимизации, представленный в этой главе, закладывает основу для оптимизации компьютерного интеллекта в алгоритмах машинного обучения, которые мы исследуем в последних главах книги.

12.1. ТЕСТИРОВАНИЕ МОДЕЛИ ЯДРА

Наша первая задача — создать модель, вычисляющую траекторию полета пушечного ядра. Модель будет реализована в виде функции на Python `trajectory`, которая принимает угол выстрела, а также несколько других параметров, которыми нам может понадобиться управлять, и возвращает список координат ядра с течением времени, пока оно не столкнется с Землей. Для построения этой модели обратимся к нашему старому знакомому из главы 9 — методу Эйлера.

Напомню, что суть метода Эйлера заключается в моделировании движения на небольших временных отрезках (мы используем отрезки, равные 0,01 с). В каждый момент мы будем получать местоположение пушечного ядра, а также его производные — скорость и ускорение. Эти величины позволяют аппроксимировать изменение положения к следующему моменту времени, и мы станем повторять этот процесс, пока пушечное ядро не упадет на землю. На каждом шаге будем сохранять время, координаты x и y ядра и возвращать их как результат функции `trajectory`.

Наконец, мы напишем функции, которые берут результаты, возвращаемые функцией `trajectory`, и оценивают одно числовое свойство. Функции `landing_position`, `hang_time` и `max_height` сообщат нам дальность выстрела, время нахождения ядра в воздухе и максимальную высоту полета соответственно. Каждая из них будет станет впоследствии предметом оптимизации.

12.1.1. Моделирование с помощью метода Эйлера

В первой попытке моделирования в двухмерном пространстве будем называть горизонтальное направление направлением x, а вертикальное — направлением z. В этом случае не придется переименовывать ни одно из них, когда мы добавим еще одно горизонтальное направление. Угол выстрела к горизонту будем обозначать θ, а скорость — $\mathbf{v}$ (рис. 12.5).

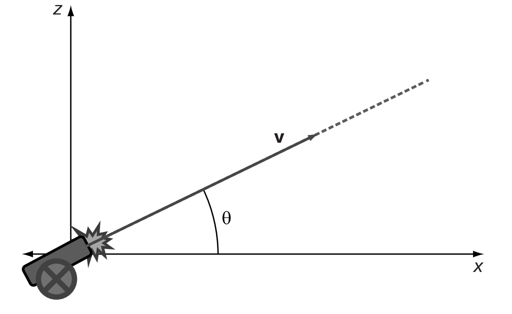


Рис. 12.5. Переменные модели полета ядра

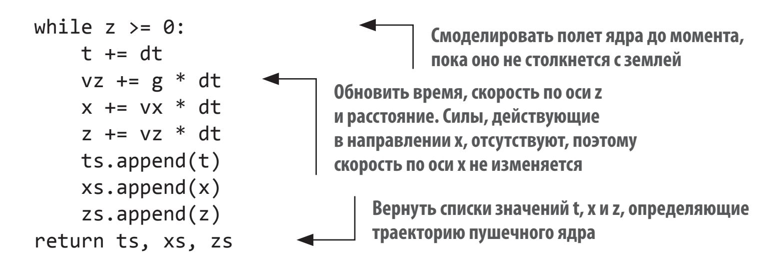
Скорость v движущегося объекта определяется как модуль его вектора скорости, то есть $v = |\mathbf{v}|$. При заданном угле выстрела θ компоненты x и z скорости полета ядра равны $v_x = |\mathbf{v}| \cdot \cos\theta$ и $v_z = |\mathbf{v}| \cdot \sin\theta$. Предполагается, что ядро покидает ствол пушки в момент времени $t = 0$ и имеет координаты $(x, z) = (0, 0)$. Дополнительно добавим возможность настройки высоты выстрела. Вот как выглядит простая реализация моделирования с использованием метода Эйлера:

Дополнительные входные параметры: временной шаг dt, напряженность гравитационного поля g и угол theta (в градусах)

```
def trajectory(theta,speed=20,height=0,
               dt=0.01,g=-9.81):
    vx = speed * cos(pi * theta / 180)
    vz = speed * sin(pi * theta / 180)
    t,x,z = 0, 0, height
    ts, xs, zs = [t], [x], [z]
```

Вычислить начальные компоненты скорости x и z, преобразовав входной угол из градусов в радианы

Инициализировать списки для сохранения значений времени и позиций x и z, вычисляемых в ходе моделирования



В примерах исходного кода для этой книги вы найдете функцию `plot_trajectories`, которая принимает результаты одного или нескольких вызовов функции `trajectory` и передает их функции `plot` из библиотеки Matplotlib, чтобы нарисовать кривые, показывающие траектории полета ядра. Например, на рис. 12.6 показаны траектории выстрелов под углами 45° и 60°, полученные следующим кодом:

```
plot_trajectories(
    trajectory(45),
    trajectory(60))
```

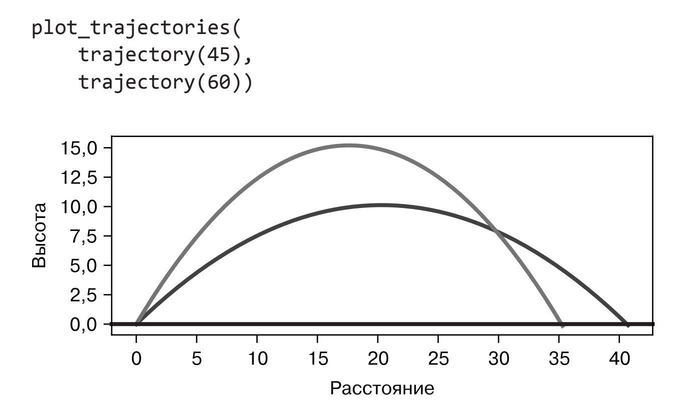


Рис. 12.6. Вывод функции plot_trajectories, показывающий траектории выстрелов под углами 45° и 60°

Мы уже можем видеть, что угол 45° дает большую дальность выстрела, а угол 60° — большую высоту. Чтобы иметь возможность оптимизировать эти свойства, нужно измерить их на множестве траекторий.

12.1.2. Измерение характеристик траектории

Конечно, полезно сохранить результаты, возвращаемые функцией `trajectory`, на случай, если понадобится построить графики их изменения, но иногда нужно сосредоточиться на одном наиболее важном числе. Примером такого числа может служить дальность выстрела — это последняя координата x в траектории перед тем, как пушечное ядро упадет на землю. Вот функция, которая получает результаты функции `trajectory` (списки с отметками времени и позициями x и z)

и возвращает дальность выстрела, или позицию приземления. Для списка traj результатов, описывающих траекторию, traj[1] содержит список координат x, а traj[1][-1] — это последний элемент в списке:

```
def landing_position(traj):
    return traj[1][-1]
```

Это основной параметр траектории, интересующий нас, но мы можем измерить и некоторые другие показатели, например, продолжительность полета ядра или его максимальную высоту. Можем написать другие функции, оценивающие эти параметры по смоделированным траекториям, например:

```
def hang_time(traj):
    return traj[0][-1]

def max_height(traj):
    return max(traj[2])
```

Общее время, в течение которого ядро находилось в воздухе, равно последнему значению времени — отметке времени, когда оно падает на землю

Максимальная высота определяется максимальным значением среди позиций z в третьем списке в результатах trajectory

Чтобы найти оптимальное значение для любого из этих показателей, нужно изучить влияние на них входных параметров, а именно угла выстрела.

12.1.3. Исследование различных углов выстрела

Функция trajectory принимает угол запуска и возвращает данные о местоположении ядра в разные моменты. Функция landing_position принимает эти данные и возвращает одно число. Объединив их (рис. 12.7), можно получить функцию дальности выстрела в зависимости от угла, которая предполагает, что все остальные параметры модели остаются неизменными.

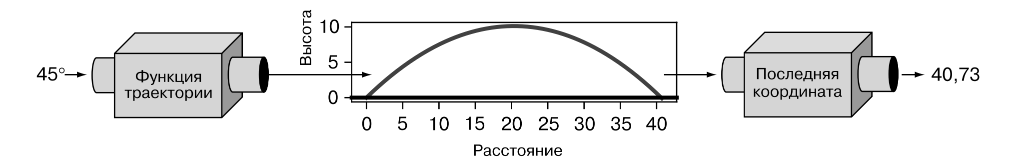


Рис. 12.7. Местоположение падения как функция угла выстрела

Один из способов проверить влияние угла выстрела на его дальность — построить график зависимости дальности от угла выстрела (рис. 12.8). Для этого нужно вычислить результат композиции landing_position(trajectory(theta)) для нескольких значений theta и передать их функции scatter из библиотеки Matplotlib. В следующем примере я использовал range(0, 95, 5) для перечисления углов выстрела. В этот диапазон входят все углы от 0 до 90° с шагом 5°:

```
import matplotlib.pyplot as plt
angles = range(0,90,5)
```

```
landing_positions = [landing_position(trajectory(theta))
                     for theta in angles]
plt.scatter(angles,landing_positions)
```

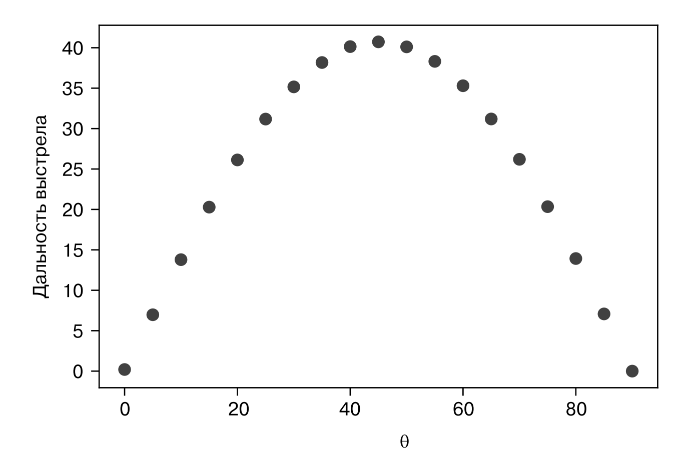


Рис. 12.8. График зависимости дальности от угла выстрела

Глядя на этот график, легко догадаться, каким будет оптимальное значение. Максимальная дальность выстрела достигается при угле 45° и составляет чуть более 40 м от точки выстрела. В данном случае 45° — это *точное* значение угла, при котором достигается максимальная дальность выстрела. В следующем разделе мы прибегнем к матанализу, чтобы без всякого моделирования подтвердить, что это действительно максимальное значение.

12.1.4. Упражнения

Упражнение 12.1. Какое расстояние пролетит пушечное ядро при выстреле под углом 50° с начальной высоты, равной нулю? А если выстрелить под углом 130°?

Решение. При выстреле под углом 50° пушечное ядро пролетит около 40,1 м в положительном направлении оси x, а при выстреле под углом 130° — 40,1 м в отрицательном направлении:

```
>>> landing_position(trajectory(50))
40.10994684444007
>>> landing_position(trajectory(130))
-40.10994684444007
```

Это объясняется тем, что угол 130° с положительным направлением оси x — это угол 50° с отрицательным направлением оси x.

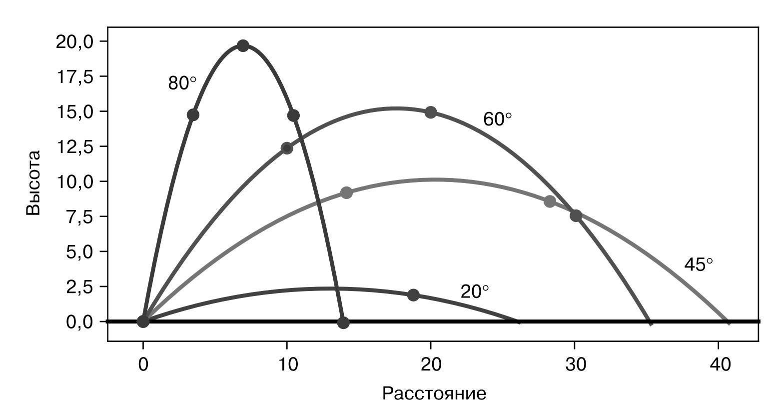
Упражнение 12.2. Мини-проект. Усовершенствуйте функцию `plot_trajectories`, добавив в нее рисование на графике траектории большой точки, соответствующей каждой полной секунде, чтобы можно было видеть течение времени на графике.

Решение. Далее приводится дополнительный код в функции. Он определяет индекс ближайшей отметки времени после каждой целой секунды и строит точечный график значений (x, z) для каждого из этих индексов:

```
def plot_trajectories(*trajs,show_seconds=False):
    for traj in trajs:
        xs, zs = traj[1], traj[2]
        plt.plot(xs,zs)
        if show_seconds:
            second_indices = []
            second = 0
            for i,t in enumerate(traj[0]):
                if t>= second:
                    second_indices.append(i)
                    second += 1
            plt.scatter([xs[i] for i in second_indices],
                        [zs[i] for i in second_indices])
...
```

Теперь можно видеть течение времени для каждой из построенных траекторий, например:

```
plot_trajectories(
    trajectory(20),
    trajectory(45),
    trajectory(60),
    trajectory(80),
    show_seconds=True)
```



Графики четырех траекторий с точками, соответствующими целому числу секунд истекшего времени

Упражнение 12.3. Постройте точечный график зависимости времени полета ядра от угла для углов от 0 до 180°. Какой угол выстрела дает максимальное время полета?

Решение:

```
test_angles = range(0,181,5)
hang_times = [hang_time(trajectory(theta)) for theta in test_angles]
plt.scatter(test_angles, hang_times)
```

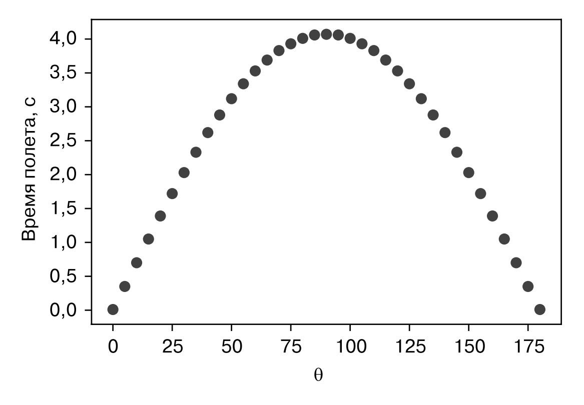


График времени полета пушечного ядра в зависимости от угла выстрела

Судя по графику, максимальное время полета ядра — около 4 с получается при выстреле под углом примерно 90°. Это интуитивно понятно, потому что $\theta = 90°$ дает начальную скорость с наибольшей вертикальной составляющей.

Упражнение 12.4. Мини-проект. Напишите функцию `plot_trajectory_metric`, отображающую график любого из имеющихся параметров по заданному набору значений угла выстрела (θ). Например, вызов

```
plot_trajectory_metric(landing_position,[10,20,30])
```

должен построить точечный график дальности выстрела в зависимости от угла для углов 10°, 20° и 30°.

Дополнительно реализуйте передачу именованных аргументов `plot_trajectory_metric` во внутренние вызовы `trajectory`, чтобы получить возможность повторно выполнить тестирование с другим параметром

моделирования. Например, следующий код выводит тот же график, но для случая, когда стрельба производится с высоты 10 м:

```
plot_trajectory_metric(landing_position,[10,20,30], height=10)
```

Решение:

```
def plot_trajectory_metric(metric,thetas,**settings):
    plt.scatter(thetas,
                [metric(trajectory(theta,**settings))
                 for theta in thetas])
```

Вот как можно построить график из предыдущего упражнения:

```
plot_trajectory_metric(hang_time, range(0,181,5))
```

Упражнение 12.5. Мини-проект. Определите приблизительный угол стрельбы с высоты 10 м, при котором дальность максимальная.

Решение. Используя функцию `plot_trajectory_metric` из предыдущего мини-проекта, можно просто выполнить

```
plot_trajectory_metric(landing_position,range(0,90,5), height=10)
```

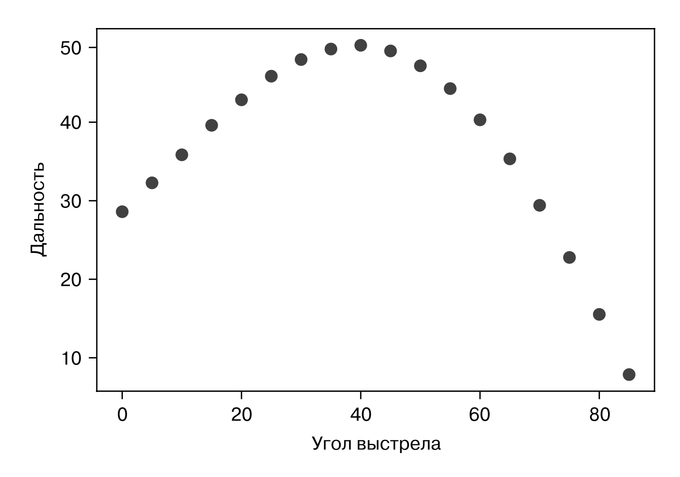


График зависимости дальности от угла выстрела с высоты 10 м

Максимальная дальность при выстреле с высоты 10 достигается при угле выстрела около 40°.

12.2. ВЫЧИСЛЕНИЕ ОПТИМАЛЬНОЙ ДАЛЬНОСТИ

Используя матанализ, можно вычислить дальность стрельбы из пушки, а также определить угол, при котором она максимальна. Во-первых, нужно придумать точную функцию, сообщающую дальность r как функцию угла выстрела θ. Хочу сразу предупредить, что это потребует некоторого погружения в алгебру. Но я проведу вас по шагам, поэтому не волнуйтесь: если вы потеряетесь, то сможете сразу перейти к окончательной форме функции $r(\theta)$ и продолжить чтение.

Затем я покажу, как использовать производные для поиска максимального значения функции $r(\theta)$ и соответствующего угла θ. В частности, значение θ, при котором производная $r'(\theta)$ равна нулю, также является значением θ, дающим максимальное значение $r(\theta)$. Кому-то может показаться непонятной эта связь, но многое прояснится, когда мы изучим график $r(\theta)$ и изменение его наклона.

12.2.1. Определение дальности в зависимости от угла выстрела

Горизонтальное расстояние, преодолеваемое пушечным ядром, довольно легко рассчитать. Компонента x скорости v_x постоянна на протяжении всего полета. За общее время полета Δt ядро преодолевает общее расстояние $r = v_x \cdot \Delta t$. Задача состоит в том, чтобы найти точное значение прошедшего времени Δt.

Это время, в свою очередь, зависит от местоположения ядра на оси z в течение полета, которое является функцией $z(t)$. Предполагается, что ядром выстрелили с начальной высоты, равной нулю, то есть в начальный момент $t = 0$ значение $z(t) = 0$. Конечный момент определяет время, прошедшее с момента выстрела. На рис. 12.9 показан график $z(t)$ для модели с $\theta = 45°$. Обратите внимание на то, что его форма очень похожа на траекторию, но горизонтальная ось (t) теперь представляет время:

```
trj = trajectory(45)
ts, zs = trj[0], trj[2]
plt.plot(ts,zs)
```

Мы знаем, что $z''(t) = g = -9{,}81$ — ускорение свободного падения. Мы также знаем начальную скорость z, $z'(0) = |\mathbf{v}| \cdot \sin\theta$ и начальное положение z, $z(0) = 0$. Чтобы восстановить функцию местоположения $z(t)$, нужно дважды проинтегрировать ускорение $z''(t)$. Первый интеграл даст скорость:

$$z'(t) = z'(0) + \int_0^t g\,d\tau = |\mathbf{v}| \cdot \sin\theta + gt.$$

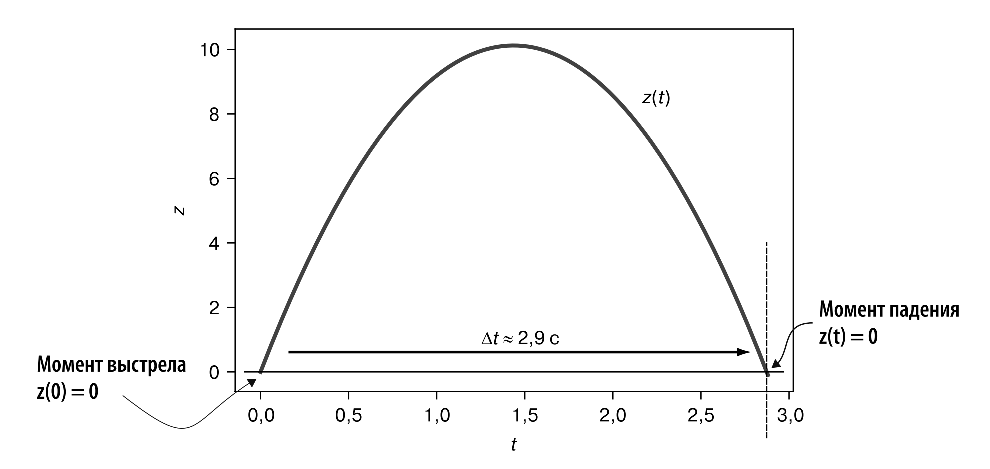


Рис. 12.9. График *z*(*t*) для ядра, показывающий моменты выстрела и падения, когда *z* = 0. На нем видно, что время полета составляет около 2,9 с

Второй интеграл даст местоположение:

$$z(t) = z(0) + \int_0^t z'(\tau)\,d\tau = \int_0^t |\mathbf{v}| \cdot \sin\theta + g\tau\,d\tau = |\mathbf{v}| \cdot \sin\theta \cdot t + \frac{g}{2}t^2.$$

Мы можем подтвердить соответствие этой формулы нашей модели, построив ее график (рис. 12.10). Он почти неотличим от графика модели:

```
def z(t):
    return 20*sin(45*pi/180)*t + (-9.81/2)*t**2

plot_function(z,0,2.9)
```

Прямой перевод результата интеграла z(t) в код на Python

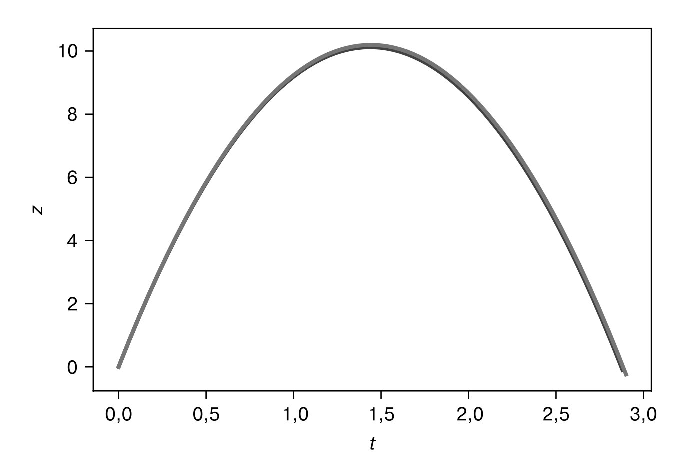


Рис. 12.10. График точной функции *z*(*t*) поверх графика модели

Для простоты запишем начальную скорость $|\mathbf{v}| \cdot \sin\theta$ как v_z, тогда $z(t) = v_z t + gt^2/2$. Нужно найти значение t, при котором $z(t) = 0$, то есть общую продолжительность полета пушечного ядра. Возможно, вы помните, как найти это значение, из школьного курса алгебры, а если нет, то прямо сейчас я кратко напомню. Чтобы узнать, какое значение t является решением уравнения $at^2 + bt + c = 0$, достаточно подставить значения a, b и c в *формулу определения корней квадратного уравнения*

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Уравнение вида $at^2 + bt + c = 0$ имеет два корня — два момента времени, когда ядро оказывается в точке с координатой $z = 0$. Символ «±» — это сокращение, показывающее, что использование знаков «+» или «–» в этом месте уравнения дает два разных, но действительных ответа.

В случае уравнения $z(t) = v_z t + gt^2/2 = 0$ мы имеем $a = g/2$, $b = v_z$ и $c = 0$. Подставляем эти значения в формулу и находим

$$t = \frac{-v_z \pm \sqrt{v_x^2}}{g} = \frac{-v_z \pm v_z}{g}.$$

Подставляя «+» вместо символа «±», получаем $t = (-v_z + v_z)/g = 0$. Этот результат говорит о том, что $z = 0$ при $t = 0$, и является хорошей проверкой пригодности формулы: он подтверждает, что пушечное ядро начало полет в точке с $z = 0$. Интересный результат получается при замене «±» знаком «–». В этом случае получается $t = (-v_z - v_z)/g = -2v_z/g$.

Попробуем подтвердить осмысленность этого результата. При начальной скорости 20 м/с и угле выстрела 45° (эти же параметры мы использовали при моделировании) момент времени пересечения оси z равен $-2 \cdot (20 \cdot \sin 45°)/-9{,}81 \approx$ $\approx 2{,}88$ с. Это близко к результату 2,9 с, который читается на графике.

Такой результат дает нам уверенность в правильности вычисленного времени полета Δt как $\Delta t = -2v_z/g$ или $\Delta t = -2|\mathbf{v}| \sin\theta/g$. Дальность выстрела равна $r = v_x \cdot \Delta t = |\mathbf{v}| \cos\theta \cdot \Delta t$, отсюда полная формула дальности r как функции угла выстрела θ имеет вид

$$r(\theta) = -\frac{2|\mathbf{v}|^2}{g} \sin\theta \cos\theta.$$

Мы можем построить график этой формулы поверх графика смоделированных позиций падения ядра при выстреле под разными углами (рис. 12.11) и убедиться, что они совпадают:

```
def r(theta):
    return (-2*20*20/-9.81)*sin(theta*pi/180)*cos(theta*pi/180)

plot_function(r,0,90)
```

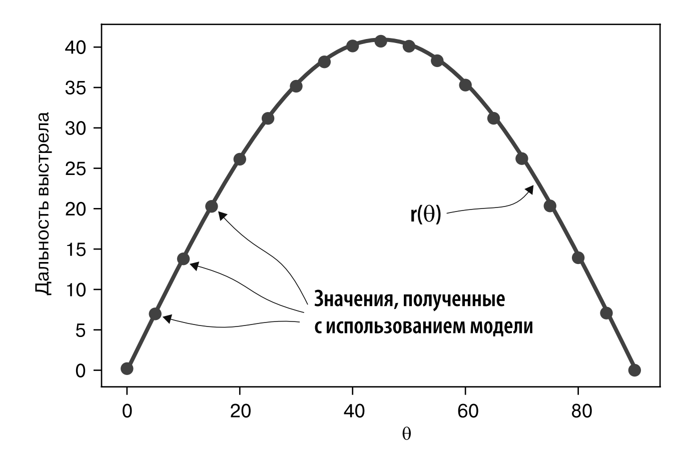


Рис. 12.11. График расчетной дальности полета ядра $r(\theta)$ в зависимости от угла выстрела соответствует дальностям, полученным с использованием модели

Наличие функции $r(\theta)$ обеспечивает большое преимущество перед многократным запуском симулятора. Во-первых, формула сообщает дальность выстрела при *любом* угле выстрела, а не только при нескольких, которые мы смоделировали. Во-вторых, вычисление одной функции требует гораздо меньше вычислительных ресурсов, чем выполнение сотен итераций методом Эйлера. Для более сложных моделей это может иметь большое значение. Кроме того, функция дает точный, а не приближенный результат. И последнее преимущество, которым воспользуемся в дальнейшем, заключается в том, что функция $r(\theta)$ гладкая, соответственно, мы сможем находить ее производные. Это поможет получить представление о том, как изменяется дальность полета ядра в зависимости от угла выстрела.

12.2.2. Решение для вычисления максимальной дальности

Глядя на график $r(\theta)$ на рис. 12.12, можно определить, как будет выглядеть производная $r'(\theta)$. С увеличением угла выстрела, начиная с нуля, дальность также увеличивается до определенного предела, но при уменьшающейся скорости. В конце концов увеличение угла выстрела начинает уменьшать дальность.

Ключевое наблюдение заключается в том, что пока $r'(\theta)$ больше нуля, дальность увеличивается с увеличением θ. Затем производная $r'(\theta)$ становится меньше нуля, и с этого момента дальность выстрела начинает уменьшаться. Именно при этом угле (когда производная равна нулю) функция $r(\theta)$ достигает максимального значения. Это можно увидеть на графике $r(\theta)$ на рис. 12.12: когда кривая достигает максимума, наклон касательной к графику становится равным нулю.

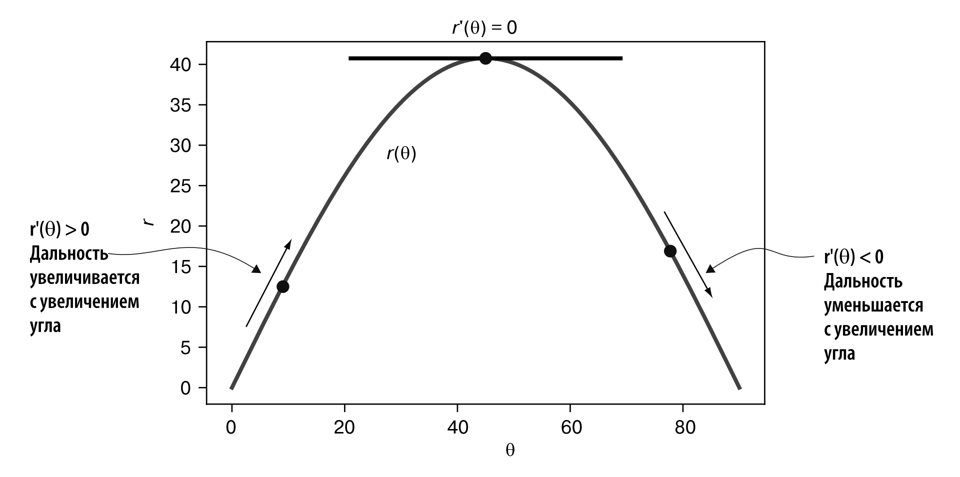


Рис. 12.12. График $r(\theta)$ достигает максимума, когда производная становится равной нулю, соответственно, наклон касательной к графику тоже равен нулю

Мы можем взять производную от $r(\theta)$ символически и найти, при каком значении θ от 0 до 90° она равна нулю, этот результат должен согласовываться с примерно определенным максимальным значением 45°. Напомню, что формула r имеет вид

$$r(\theta) = -\frac{2|\mathbf{v}|^2}{g}\sin\theta\cos\theta.$$

Поскольку $-2|\mathbf{v}|^2/g$ не зависит от θ, единственной сложностью оказывается применение правила произведения для sin θ cos θ. Результат

$$r'(\theta) = \frac{2|\mathbf{v}|^2}{g}\left(\cos^2\theta - \sin^2\theta\right).$$

Обратите внимание на то, что я убрал знак «минус». Для тех, кто прежде не видел обозначения $\sin^2\theta$, поясню, что оно означает $(\sin\theta)^2$. Значение производной $r'(\theta)$ равно нулю, когда выражение $\sin^2\theta - \cos^2\theta$ равно нулю (проще говоря, мы можем игнорировать константу $2|\mathbf{v}|^2/g$). Выяснить значение θ, при котором это выражение равно нулю, можно несколькими способами, но особенно хорошим оказывается использование тригонометрического тождества $\cos 2\theta = \cos^2\theta - \sin^2\theta$, что еще больше упрощает нашу задачу. Теперь нужно выяснить, при каком значении θ верно равенство $\cos 2\theta = 0$.

Функция косинуса равна нулю при $\pi/2$ плюс любое кратное π, или 90° плюс любое кратное 180°, то есть 90°, 270°, 430° и т. д. Если 2θ равно этим значениям, то θ может быть равным половине любого из этих значений: 45°, 135°, 215° и т. д. Из них наиболее интересны два значения. Первое, θ = 45° — это решение в диапазоне от θ = 0 до θ = 90°, то есть то самое решение, которое мы ищем! Второе интересное решение — 135°, потому что это то же самое, что выстрелить под углом 45° в противоположном направлении (рис. 12.13).

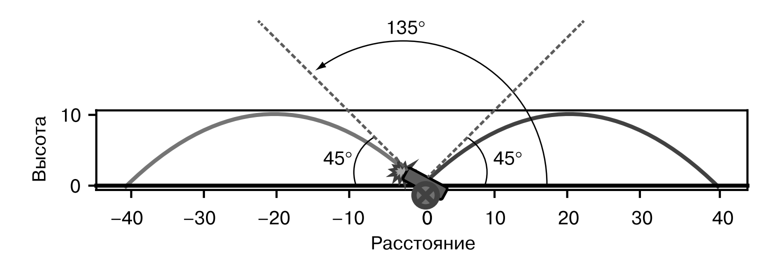


Рис. 12.13. В нашей модели выстрел под углом 135° подобен выстрелу под углом 45°, только в противоположном направлении

При углах 45° и 135° дальность выстрела равна

```
>>> r(45)
40.774719673802245
>>> r(135)
-40.77471967380224
```

Это максимальные расстояния, на которые может улететь пушечное ядро при прочих равных параметрах. Угол выстрела 45° обеспечивает максимальное удаление точки падения, а угол запуска 135° — минимальное удаление.

12.2.3. Идентификация максимумов и минимумов

Чтобы увидеть разницу между максимальной дальностью при выстреле под углом 45° и минимальной — при выстреле под углом 135°, расширим график $r(\theta)$. Напомню, что оба этих угла соответствуют нулевому значению производной $r'(\theta)$, как показано на рис. 12.14.

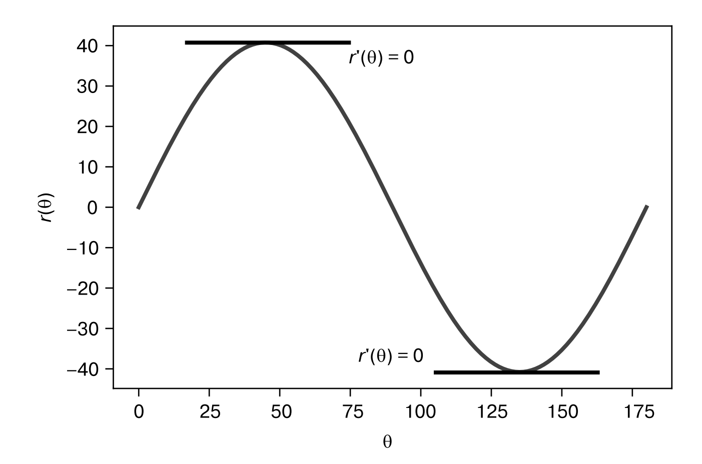


Рис. 12.14. Углы $\theta = 45°$ и $\theta = 135°$ — это два значения в диапазоне от 0 до 180°, при которых $r'(\theta) = 0$

Хотя *максимумы* гладких функций находятся там, где производная равна нулю, обратное утверждение не всегда верно — не каждая точка на графике функции, где производная равна нулю, является максимальным значением. Как можно видеть на рис. 12.14, при $\theta = 135°$ функция имеет *минимальное* значение.

Следует также быть осторожными с глобальным поведением функций, потому что производная может быть равна нулю в так называемом *локальном* максимуме или минимуме, когда значение функции оказывается максимальным или минимальным в пределах ограниченной области определений, а истинные глобальные максимальное или минимальное значения находятся в другом месте. На рис. 12.15 показан классический пример — $y = x^3 - x$. В ограниченной области $-1 < x < 1$ есть две точки, где производная равна нулю и которые выглядят как максимум и минимум. Но если раздвинуть рамки обозреваемой области, можно увидеть, что ни одно из этих значений не является максимальным или минимальным для всей функции, потому что ее ветви уходят в бесконечность в обоих направлениях.

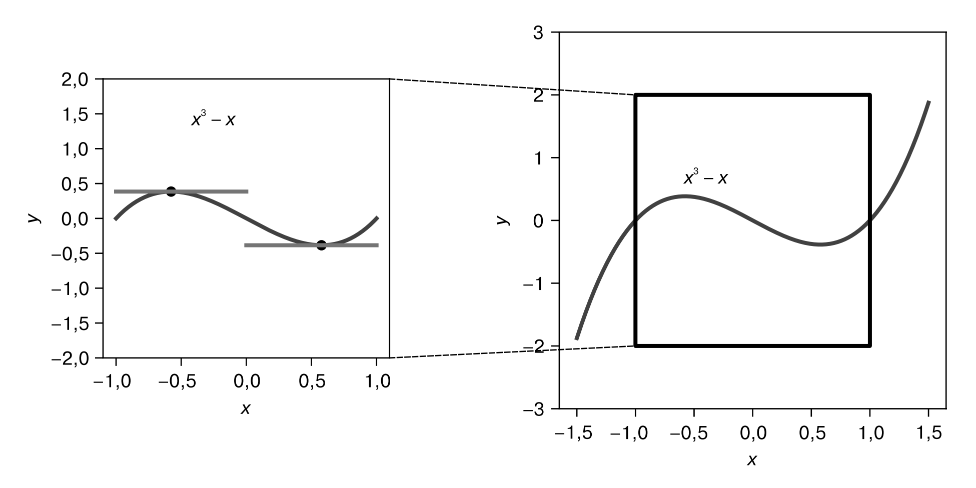


Рис. 12.15. Две точки, которые являются локальным минимумом и локальным максимумом, но не являются ни минимальным, ни максимальным значением функции

Еще один источник путаницы заключается в том, что точка, в которой производная равна нулю, может даже не быть локальным минимумом или максимумом. Например, функция $y = x^3$ имеет нулевую производную при $x = 0$ (рис. 12.16). В этой точке функция x^3 на мгновение перестает возрастать.

Я не буду вдаваться в технические детали того, как определить, является ли точка с нулевой производной минимумом, максимумом или ни тем ни другим,

или как отличить локальные минимумы и максимумы от глобальных. Отмечу лишь, что основная суть заключается в том, что всегда необходимо оценивать полное поведение функции, прежде чем заявлять о том, что найдено оптимальное значение. Помня об этом, перейдем к некоторым более сложным функциям и методам их оптимизации.

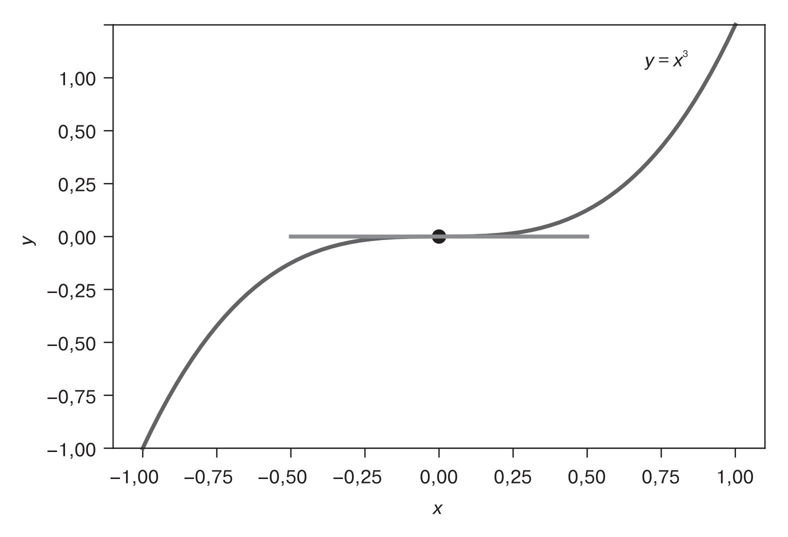


Рис. 12.16. Производная функции $y = x^3$ равна нулю при $x = 0$, но это не минимальное и не максимальное значение

12.2.4. Упражнения

Упражнение 12.6. Используйте формулу определения времени Δt полета ядра при угле выстрела θ, чтобы найти угол, при котором ядро находится в воздухе максимальное время.

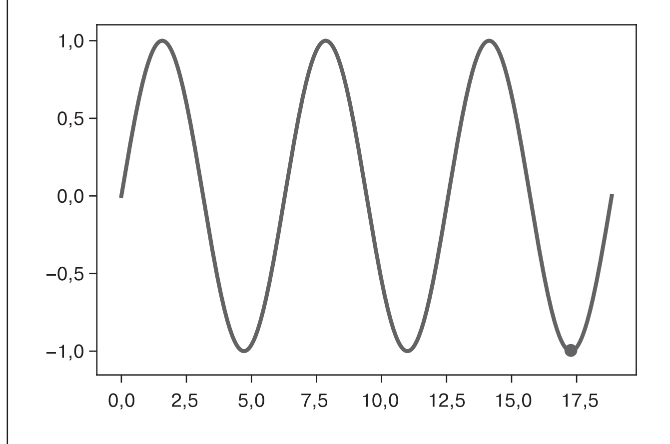
Решение. Время полета ядра $t = 2v_z/g = 2v \sin \theta/g$, где начальная скорость ядра $v = |\mathbf{v}|$. Максимальным время становится, когда $\sin \theta$ имеет максимальное значение. Чтобы определить это значение, не нужно прибегать к вычислениям, потому что максимальное значение $\sin \theta$ в диапазоне $0 \leq \theta \leq 180^\circ$ получается при $\theta = 90^\circ$. Иначе говоря, при неизменности всех остальных параметров ядро будет оставаться в воздухе дольше всего, когда оно выпущено вертикально вверх.

Упражнение 12.7. Проверьте, действительно ли производная sin (x) равна нулю при $x = 11\pi/2$. Это максимальное или минимальное значение sin (x)?

Решение. Производная от sin (x) есть cos (x), отсюда

$$\cos\left(\frac{11\pi}{2}\right) = \cos\left(\frac{3\pi}{2} + 4\pi\right) = \cos\left(\frac{3\pi}{2}\right) = 0,$$

то есть производная sin (x) действительно равна нулю при $x = 11\pi/2$. Поскольку $\sin(11\pi/2) = \sin(3\pi/2) = -1$, а функция синуса изменяется в диапазоне от -1 до 1, можно с уверенностью сказать, что это локальный минимум. Вот график sin (x), подтверждающий это.



Упражнение 12.8. Определите, при каких значениях x функция $f(x) = x^3 - x$ имеет локальный минимум и локальный максимум. Какие значения она получает при этом?

Решение. Из графика функции видно, что $f(x)$ достигает локального минимума при некотором $x > 0$ и локального максимума при некотором $x < 0$. Найдем эти две точки.

Производная имеет вид $f'(x) = 3x^2 - 1$, поэтому нужно найти точки, в которых $3x^2 - 1 = 0$. Задачу можно было бы решить как квадратное уравнение, но в данном случае решение вполне очевидно: если $3x^2 - 1 = 0$, то $x^2 = 1/3$, поэтому $x = -1/\sqrt{3}$ или $x = 1/\sqrt{3}$. Это значения x, при которых $f(x)$ достигает своего локального минимума и максимума.

Локальное максимальное значение

$$f\left(\frac{-1}{\sqrt{3}}\right)=\frac{-1}{3\sqrt{3}}-\frac{-1}{\sqrt{3}}=\frac{2}{3\sqrt{3}},$$

локальное минимальное значение

$$f\left(\frac{1}{\sqrt{3}}\right)=\frac{1}{3\sqrt{3}}-\frac{1}{\sqrt{3}}=\frac{-2}{3\sqrt{3}}.$$

Упражнение 12.9. Мини-проект. График квадратичной функции $q(x) = ax^2 + bx + c$ при а $\neq 0$ представляет собой *параболу*, которая имеет либо одно максимальное значение, либо одно минимальное значение. Основываясь на значениях a, b и c, определите, при каком значении x функция $q(x)$ достигает максимума или минимума. Как узнать, является эта точка минимумом или максимумом?

Решение. Производная $q'(x)$ определяется как $2ax + b$. Она равна нулю при $x = -b/2a$.

Если a положительно, производная получает отрицательное значение при некотором малом значении x, затем достигает нуля при $x = -b/2a$ и после этого становится положительной. Это означает, что q уменьшается до $x = -b/2a$, а затем увеличивается, следовательно, в этом случае нулевая производная соответствует минимальному значению $q(x)$.

И наоборот, если a отрицательно. Таким образом, $x = -b/2a$ является минимальным значением $q(x)$, если a положительно, и максимальным, если a отрицательно.

12.3. УСОВЕРШЕНСТВОВАНИЕ МОДЕЛИ

Усложнив модель, мы можем добавить возможность управлять ее поведением с использованием нескольких параметров. Для исходной модели пушки угол выстрела θ был единственным параметром, с которым мы экспериментировали. Для оптимизации дальности стрельбы применили функцию одной переменной $r(\theta)$. В этом разделе мы реализуем модель стрельбы из пушки в трехмерном пространстве, то есть в качестве параметров будут использоваться два угла выстрела, которые нужно будет оптимизировать, чтобы получить максимальную дальность полета пушечного ядра.

12.3.1. Добавление еще одного измерения

Прежде всего добавим в нашу модель ось y. Теперь можно изобразить пушку стоящей в начале координат на плоскости xy и стреляющей ядром вверх, в направлении z, под некоторым углом θ. В этой версии мы можем управлять углом θ, а также еще одним углом, который назовем ϕ (греческая буква «фи»). Он показывает, насколько пушка повернута вбок от направления $+x$ (рис. 12.17).

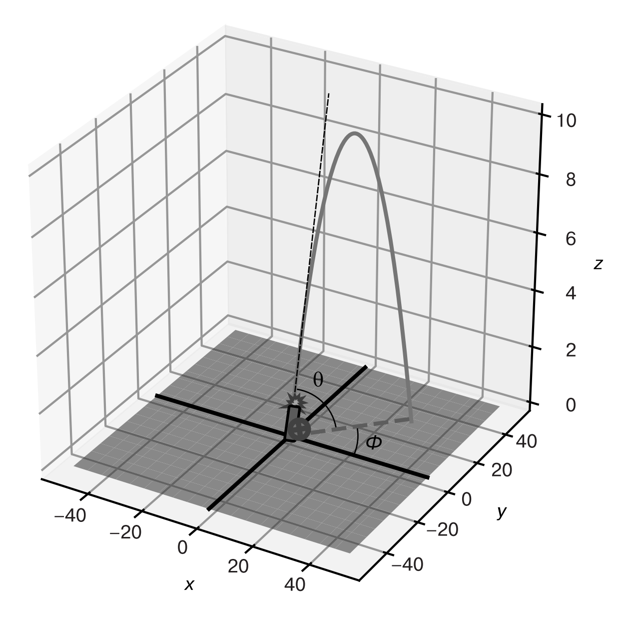


Рис. 12.17. Стрельба из пушки в трехмерном пространстве. Два угла, θ и ϕ, определяют направление выстрела из пушки

Для моделирования стрельбы из пушки в трехмерном пространстве нужно добавить движение в направлении y. Физика в направлении z остается точно такой же, но горизонтальная скорость делится между направлениями x и y в зависимости от значения угла ϕ. Если прежде составляющая x начальной скорости была $v_x = |\mathbf{v}| \cos \theta$, то теперь она умножается на коэффициент $\cos \phi$, что дает уравнение $v_x = |\mathbf{v}| \cos \theta \cos \phi$. Составляющая y начальной скорости равна $v_y = |\mathbf{v}| \cos \theta \sin \phi$. Поскольку вдоль направления y гравитация не действует, нам не нужно обновлять v_y в процессе моделирования. Вот обновленная функция траектории:

```
def trajectory3d(theta,phi,speed=20,
                 height=0,dt=0.01,g=-9.81):    ← Угол φ поворота вбок является входным параметром модели
    vx = speed * cos(pi*theta/180)*cos(pi*phi/180)
    vy = speed * cos(pi*theta/180)*sin(pi*phi/180)    ← Вычислить начальную скорость вдоль оси y
    vz = speed * sin(pi*theta/180)
    t,x,y,z = 0, 0, 0, height
```

```
    ts, xs, ys, zs = [t], [x], [y], [z]   ◄── Значения времени и координаты
    while z >= 0:                              x и y будут сохраняться на
        t += dt                                протяжении всего периода
        vz += g * dt                           моделирования
        x += vx * dt
        y += vy * dt   ◄── Обновить координаты
        z += vz * dt       в каждой итерации
        ts.append(t)
        xs.append(x)
        ys.append(y)
        zs.append(z)
    return ts, xs, ys, zs
```

В этой модели угол θ, обеспечивающий максимальную дальность стрельбы, остался прежним. Независимо от направления в плоскости — $+x$, $-x$ или любого другого — ядро, выпущенное под углом 45° к горизонту, должно пролететь одинаковое расстояние. То есть ϕ не влияет на дальность стрельбы. Теперь добавим рельеф на местности с переменной высотой вокруг точки запуска, чтобы повлиять на дальность стрельбы.

12.3.2. Моделирование рельефа местности вокруг пушки

При наличии холмов и долин вокруг пушки ядро может оставаться в воздухе разное время в зависимости от направления выстрела. Смоделировать возвышение или понижение плоскости $z = 0$ можно с помощью функции, возвращающей число для каждой точки (x, y). Например,

```
def flat_ground(x,y):
    return 0
```

представляет плоскую поверхность, высота каждой точки (x, y) которой равна нулю. Другая функция, которую мы используем, — это хребет между двумя долинами:

```
def ridge(x,y):
    return (x**2 - 5*y**2) / 2500
```

Линия хребта постепенно повышается от начала координат в положительном и отрицательном направлениях вдоль оси x, образуя седловину, а склоны спускаются в положительном и отрицательном направлениях вдоль оси y. (Можете построить сечения этой функции при $x = 0$ и $y = 0$, чтобы подтвердить это.)

Независимо от того, на каком рельефе будет моделироваться полет ядра, на плоской поверхности или на хребте, мы должны адаптировать функцию `trajectory3d`, чтобы она завершалась с падением ядра на землю, а не когда его высота достигнет нуля. Для этого можно передать функцию, определяющую рельеф в именованном аргументе, по умолчанию принимающем функцию

flat_ground, и изменить проверку момента падения ядра на землю. Вот измененные строки в функции:

```
def trajectory3d(theta,phi,speed=20,height=0,dt=0.01,g=-9.81,
                 elevation=flat_ground):
    ...
    while z >= elevation(x,y):
        ...
```

В примерах исходного кода вы найдете также функцию plot_trajectories_3d, которая отображает траекторию, вычисленную функцией trajectory3D, и заданный ландшафт. В подтверждение результатов моделирования на рис. 12.18 показано, что конец траектории оказывается ниже $z = 0$ при стрельбе в сторону снижения склона и выше $z = 0$ при стрельбе в сторону его возвышения:

```
plot_trajectories_3d(
    trajectory3d(20,0,elevation=ridge),
    trajectory3d(20,270,elevation=ridge),
    bounds=[0,40,-40,0],
    elevation=ridge)
```

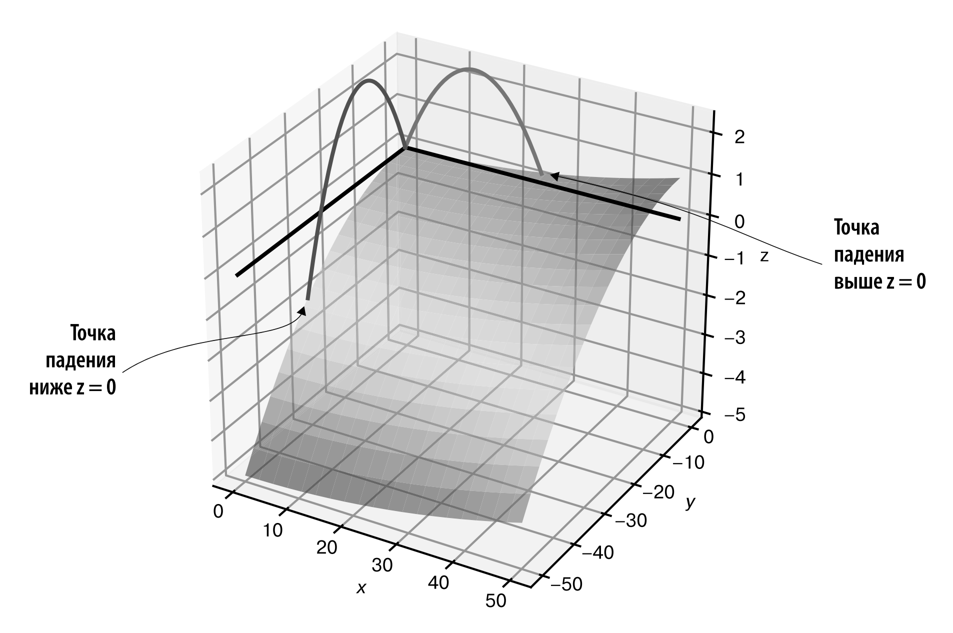


Рис. 12.18. Ядро, выпущенное вниз по склону, приземляется ниже $z = 0$, а ядро, выпущенное вверх по склону, приземляется выше $z = 0$

Нетрудно догадаться, что максимальная дальность достигается при стрельбе вниз по склону, а не вверх, потому что в этом случае пушечное ядро должно падать дольше и, соответственно, лететь дальше. Но пока неясно, даст ли угол $\theta = 45^\circ$

максимальную дальность, потому что до этого в своих расчетах мы полагали, что поверхность земли плоская. Чтобы ответить на этот вопрос, нужно записать дальность полета ядра r как функцию от θ и ϕ.

12.3.3. Решение для вычисления дальности стрельбы в трехмерном пространстве

В нашей последней модели стрельба производится в трехмерном пространстве, однако траектория полета ядра лежит в вертикальной плоскости. Соответственно, при заданном угле ϕ вычисления должны производиться в пределах среза местности в направлении выстрела. Например, если пушечное ядро выпущено под углом $\phi = 240°$, то нам нужно знать только высоту точек (x, y), лежащих на линии, исходящей из начала координат и образующей угол 240° с положительным направлением оси x. Об этой линии можно думать как о тени, отбрасываемой траекторией (рис. 12.19).

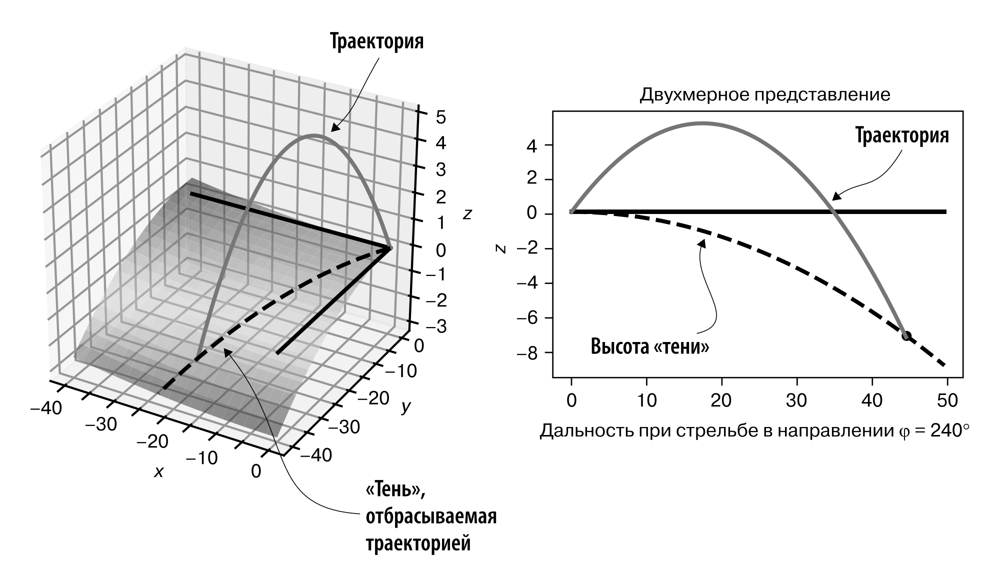


Рис. 12.19. Нужно знать только высоту точек на местности, расположенных вдоль линии выстрела. Эту линию можно представить как тень, отбрасываемую траекторией

Наша цель — выполнить все вычисления в плоскости траектории, работая с расстоянием d от точки выстрела в плоскости xy в качестве системы координат, а не с самими координатами x и y. На некотором расстоянии, в точке падения ядра, его траектория и высота местности будут иметь одинаковые значения z. Это расстояние и есть дальность выстрела, и для него нужно найти выражение.

Продолжим обозначать высоту траектории z. С течением времени высота меняется точно так же, как в двухмерной модели:

$$z(t) = v_z \cdot t + \frac{1}{2}gt^2,$$

где v_z — компонента z начальной скорости. Координаты x и y тоже задаются как простые функции времени $x(t) = v_x t$ и $y(t) = v_y t$, поскольку в направлениях x и y не действуют никакие силы.

Высота хребта задается как функция координат x и y: $(x^2 - 5y^2)/2500$. Эту высоту можно записать как $h(x, y) = Bx^2 - Cy^2$, где $B = 1/2500 = 0{,}0004$ и $C = 5/2500 = 0{,}002$. Полезно также знать высоту местности непосредственно под ядром в данный момент времени t, которую можем обозначить $h(t)$. Значение h можно вычислить в любой момент t, потому что местоположение ядра по осям x и y задается выражениями $v_x t$ и $v_y t$, а высота в той же точке (x, y) будет $h(v_x t, v_y t) = Bv_x{}^2t^2 - Cv_y{}^2t^2$.

Высота ядра над землей в момент времени t определяется как разность между $z(t)$ и $h(t)$. Время падения — это момент, когда разница оказывается равной нулю, то есть $z(t) - h(t) = 0$. Мы можем развернуть это условие, подставив определения $z(t)$ и $h(t)$:

$$\left(v_z \cdot t + \frac{1}{2}gt^2\right) - \left(Bv_x^2t^2 - Cv_y^2t^2\right) = 0.$$

И снова можем преобразовать это уравнение в форму $at^2 + bt + c = 0$:

$$\left(\frac{g}{2} - Bv_x^2 + Cv_y^2\right)t^2 - v_z t = 0.$$

В частности, $a = \frac{g}{2} - Bv_x^2 + Cv_y^2$, $b = v_z$ и $c = 0$. Чтобы найти время t, удовлетворяющее этому уравнению, можем использовать квадратичную формулу

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Поскольку $c = 0$, формула упрощается до

$$t = \frac{-b \pm b}{2a}.$$

При подстановке оператора «+» мы получаем $t = 0$. Это подтверждает, что в момент выстрела пушечное ядро находится на уровне земли. При подстановке оператора «–» получается более интересное решение, обозначающее время

падения ядра. Это время $t = (-b - b)/2a = -b/a$. Подставляя выражения для a и b, получаем выражение в терминах величин, которые мы умеем вычислять, дающее время полета ядра:

$$t = \frac{-v_z}{\frac{g}{2} - Bv_x^2 + Cv_y^2}.$$

Дальность выстрела в плоскости xy для этого времени t равна $\sqrt{x(t)^2 + y(t)^2}$. После подстановки выражений получаем: $\sqrt{(v_x t)^2 + (v_y t)^2} = t\sqrt{v_x^2 + v_y^2}$. Член $\sqrt{v_x^2 + v_y^2}$ можно рассматривать как составляющую начальной скорости на плоскости xy, поэтому я назову это число v_{xy}. Дальность полета ядра

$$d = \frac{-v_z \cdot v_{xy}}{\frac{g}{2} - Bv_x^2 + Cv_y^2}.$$

Все члены выражения либо являются константами, которые я указал, либо вычисляются через начальную скорость $v = |\mathbf{v}|$ и углы стрельбы θ и ϕ. Эти формулы можно, хотя и не без некоторых усилий, перевести на язык Python, и тогда станет ясно, что расстояние можно интерпретировать как функцию θ и ϕ:

```
B = 0.0004        # Константы, описывающие форму хребта,
C = 0.005         # начальную скорость ядра и ускорение
v = 20            # свободного падения
g = -9.81

def velocity_components(v,theta,phi):        # Вспомогательная функция,
    vx = v * cos(theta*pi/180) * cos(phi*pi/180)   # вычисляющая составляющие
    vy = v * cos(theta*pi/180) * sin(phi*pi/180)   # начальной скорости по осям x, y и z
    vz = v * sin(theta*pi/180)
    return vx,vy,vz

def landing_distance(theta,phi):
    vx, vy, vz = velocity_components(v, theta, phi)
    v_xy = sqrt(vx**2 + vy**2)      # Горизонтальная составляющая начальной скорости (параллельно плоскости xy)
    a = (g/2) - B * vx**2 + C * vy**2      # Константы a и b
    b = vz
    landing_time = -b/a      # Решение квадратичного уравнения относительно времени полета, которое равно –b/a
    landing_distance = v_xy * landing_time      # Горизонтальная дальность выстрела
    return landing_distance
```

Горизонтальная дальность выстрела равна произведению горизонтальной скорости на прошедшее время. Нанеся эту точку рядом со смоделированной траекторией, можно убедиться, что вычисленное значение точки падения ядра соответствует смоделированному методом Эйлера (рис. 12.20).

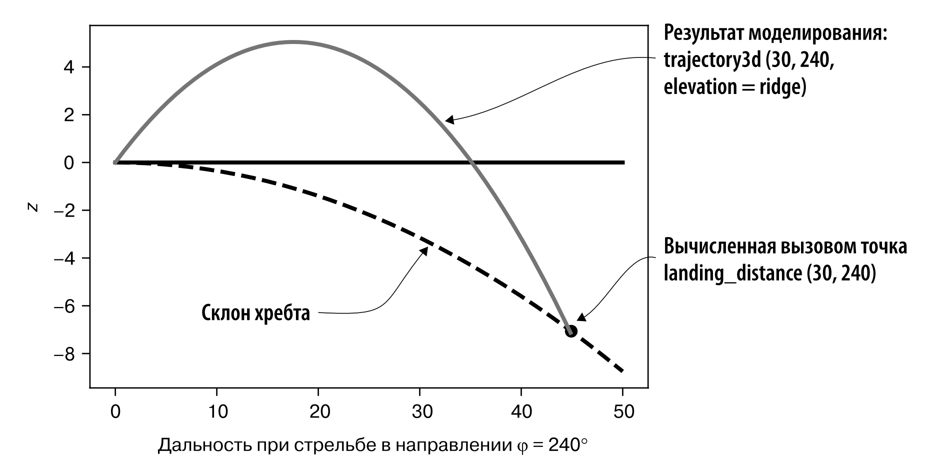


Рис. 12.20. Сравнения вычисленной точки падения ядра со смоделированной для θ = 30° и ϕ = 240°

Теперь, имея функцию $r(\theta, \phi)$ вычисления дальности стрельбы для углов θ и ϕ, можно перейти к задаче поиска углов, оптимизирующих дальность.

12.3.4. Упражнения

Упражнение 12.10. Пусть $|\mathbf{v}| = v$ — начальная скорость пушечного ядра. Докажите, что вектор начальной скорости имеет модуль, равный v. Иначе говоря, покажите, что вектор $(v \cos\theta \cos\phi, v \cos\theta \sin\phi, v \sin\theta)$ имеет длину v.

Подсказка. По определениям синуса и косинуса и теореме Пифагора $\sin^2 x + \cos^2 x = 0$ для любого значения x.

Решение. Величина $(v \cos\theta \cos\phi, v \cos\theta \sin\phi, v \sin\theta)$ определяется выражением

$$\sqrt{v^2\cos^2\theta\cos^2\varphi + v^2\cos^2\theta\sin^2\varphi + v^2\sin^2\theta} =$$

$$= \sqrt{v^2\left(\cos^2\theta\cos^2\varphi + \cos^2\theta\sin^2\varphi + \sin^2\theta\right)} =$$

$$= \sqrt{v^2\left(\cos^2\theta\left(\cos^2\varphi + \sin^2\varphi\right) + \sin^2\theta\right)} =$$

$$= \sqrt{v^2\cos^2\theta\cdot 1 + \sin^2\theta} =$$

$$= \sqrt{v^2\cdot 1} =$$

$$= v.$$

Упражнение 12.11. Запишите в явном виде формулу дальности стрельбы по гребню с возвышением $Bx^2 - Cy^2$ как функцию переменных θ и ϕ. В уравнении имеются константы B и C, а также начальная скорость ядра v и ускорение свободного падения g.

Решение. Начнем с формулы

$$d = \frac{-v_z \cdot v_{xy}}{\frac{g}{2} - Bv_x^2 + Cv_y^2}.$$

Подставив $v_z = v \sin\theta$, $v_{xy} = v\cos\theta$, $v_y = v\cos\theta\sin\phi$ и $v_x = v\cos\theta\cos\phi$, получаем:

$$d(\theta, \varphi) = \frac{-v^2 \sin\theta\cos\theta}{\frac{g}{2} - Bv^2\cos^2\theta\cos^2\varphi + Cv^2\cos^2\theta\sin^2\varphi}.$$

Упрощая знаменатель, приводим формулу к виду

$$d(\theta, \varphi) = \frac{-v^2 \sin\theta\cos\theta}{\frac{g}{2} - v^2\cos^2\theta \cdot \left(C\sin^2\varphi - B\cos^2\varphi\right)}.$$

Упражнение 12.12. Мини-проект. Когда объект, такой как пушечное ядро, быстро движется в воздухе, на него действует сила трения о воздух, называемая *сопротивлением* и воздействующая в направлении, противоположном движению. Сила сопротивления зависит от множества факторов, включая размер и форму ядра и плотность воздуха, но для простоты предположим, что она определяется так, как описано далее. Если v — это вектор скорости пушечного ядра в любой точке, то сила сопротивления

$$\mathbf{F}_\mathrm{d} = -\alpha v,$$

где α (греческая буква «альфа») — это число, обозначающее величину сопротивления, ощущаемого конкретным объектом в воздухе. Тот факт, что сила сопротивления пропорциональна скорости, означает, что по мере роста ускорения объект будет испытывать все большее и большее сопротивление. Выясните, как добавить параметр сопротивления в модель пушечного ядра, и покажите, что сопротивление замедляет его.

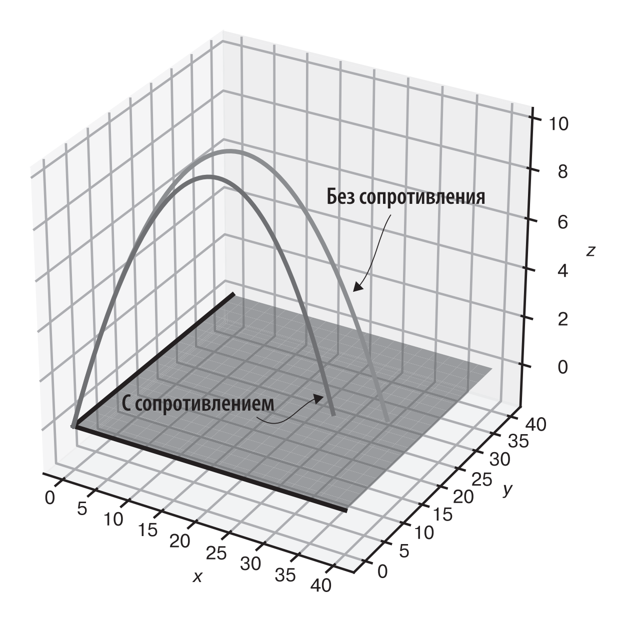
Решение. Мы должны добавить в свою модель ускорение, основанное на сопротивлении. Сила равна $-\alpha v$, а значит, вызванное ею ускорение составляет $-\alpha v/m$. Масса ядра не изменяется, поэтому можно использовать одну постоянную сопротивления α/m. Составляющие ускорения по осям, обусловленного сопротивлением, равны $v_x\alpha/m$, $v_y\alpha/m$ и $v_z\alpha/m$. Вот дополненный фрагмент кода:

```
def trajectory3d(theta,phi,speed=20,height=0,dt=0.01,g=-9.81,
                elevation=flat_ground, drag=0):
    ...
    while z >= elevation(x,y):
        t += dt
        vx -= (drag * vx) * dt
        vy -= (drag * vy) * dt
        vz += (g - (drag * vz)) * dt
        ...
    return ts, xs, ys, zs
```

Уменьшить составляющие vx и vy пропорционально силе сопротивления

Изменить скорость вдоль оси z (vz), сымитировав влияние силы гравитации и сопротивления воздуха

Здесь видно, что даже небольшая постоянная сопротивления 0,1 заметно замедляет пушечное ядро, заставляя его отклониться от траектории полета в отсутствие сопротивления.



Траектории полета пушечного ядра с drag = 0 и drag = 0,1

12.4. ОПТИМИЗАЦИЯ ДАЛЬНОСТИ С ПОМОЩЬЮ ГРАДИЕНТНОГО ВОСХОЖДЕНИЯ

Продолжим считать, что мы стреляем из пушки на местности с хребтом под некоторыми углами θ и ϕ, а остальные параметры стрельбы имеют значения по умолчанию. В данном случае функция $r(\theta, \phi)$ сообщает дальность полета ядра при этих углах стрельбы. Чтобы получить качественное представление о том, как углы влияют на дальность, можно построить график функции r.

12.4.1. График зависимости дальности от параметров стрельбы

В предыдущей главе я показал несколько способов построения графика функции двух переменных. Далее мы построим график $r(\theta, \phi)$ в виде тепловой карты. На двухмерном холсте можем изменять θ в одном направлении и ϕ в другом, а затем цветом обозначить соответствующую дальность полета ядра (рис. 12.21).

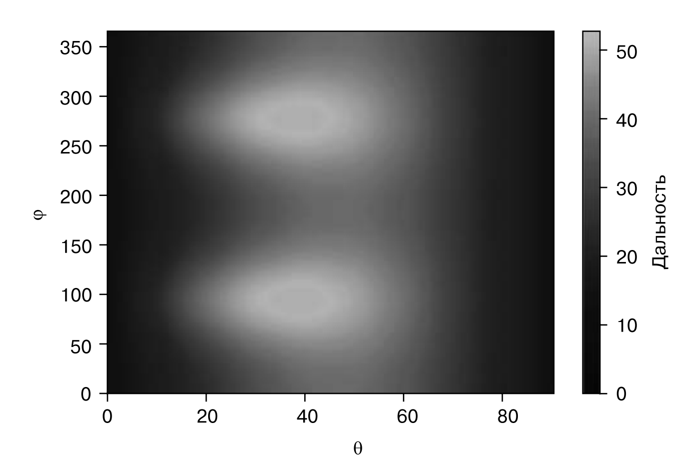


Рис. 12.21. Тепловая карта дальности стрельбы из пушки как функции углов выстрела θ и ϕ

Это абстрактное двухмерное пространство с осями координат θ и ϕ. То есть данный прямоугольник не является двухмерным срезом моделируемого трехмерного мира. Скорее, это просто удобный способ показать, как изменяется дальность r при изменении двух параметров.

На графике на рис. 12.22 более яркие области соответствуют большим дальностям, и кажется, что есть две самые яркие точки. Это возможные максимальные значения дальности стрельбы из пушки.

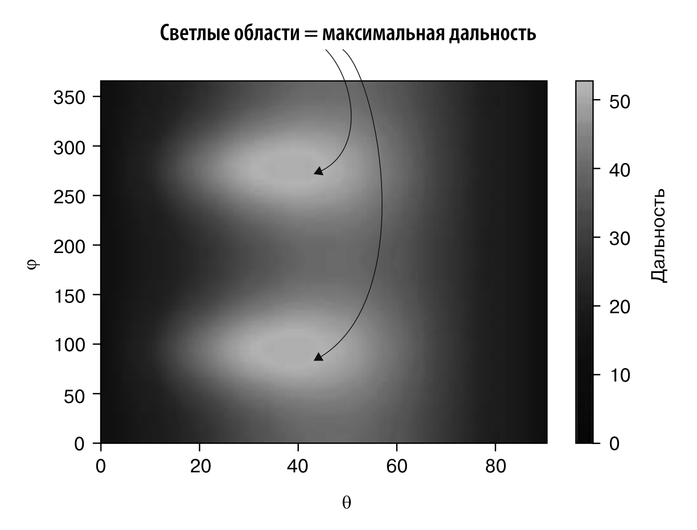


Рис. 12.22. Более яркие области соответствуют большим дальностям полета ядра

Эти точки соответствуют примерно $\theta = 40$, $\phi = 90$ и $\phi = 270$. Значения ϕ имеют определенный смысл, поскольку они представляют направления стрельбы в сторону понижения рельефа. Наша следующая цель — найти точные значения θ и ϕ, обеспечивающие максимальную дальность.

12.4.2. Градиент функции дальности

Так же как прежде мы использовали производную функции одной переменной для поиска ее максимума, задействуем градиент $\nabla r(\theta, \phi)$ для поиска максимумов функции $r(\theta, \phi)$. Как мы видели, когда гладкая функция одной переменной $f(x)$ достигает максимума, ее производная $f'(x) = 0$. В этой точке график $f(x)$ на мгновение становится горизонтальным, то есть наклон $f(x)$ равен нулю, или, точнее, наклон линии наилучшей аппроксимации в данной точке равен нулю. Точно так же, если построить трехмерный график $r(\theta, \phi)$, мы увидим, что он горизонтальный в точках максимума (рис. 12.23).

Уточним, что это значит. Поскольку $r(\theta, \phi)$ — гладкая функция, существует плоскость наилучшей аппроксимации. Наклоны этой плоскости в направлениях θ и ϕ определяются частными производными $\partial r/\partial\theta$ и $\partial r/\partial\phi$ соответственно. Только когда оба наклона равны нулю, плоскость становится горизонтальной, что означает: график $r(\theta, \phi)$ — горизонтальный.

Поскольку частные производные r определяются как компоненты градиента r, это условие горизонтальности эквивалентно утверждению, что $\nabla r(\theta, \phi) = 0$. Чтобы найти такие точки, нужно взять градиент полной формулы для $r(\theta, \phi)$,

а затем решить его относительно значений θ и ϕ, отыскав точки, в которых он равен нулю. Вычисление этих производных требует значительных усилий и не особенно познавательно, поэтому я оставлю его вам в качестве упражнения. Далее покажу способ *аппроксимации* градиента вверх по наклону графика до точки максимума, не требующий никаких математических вычислений.

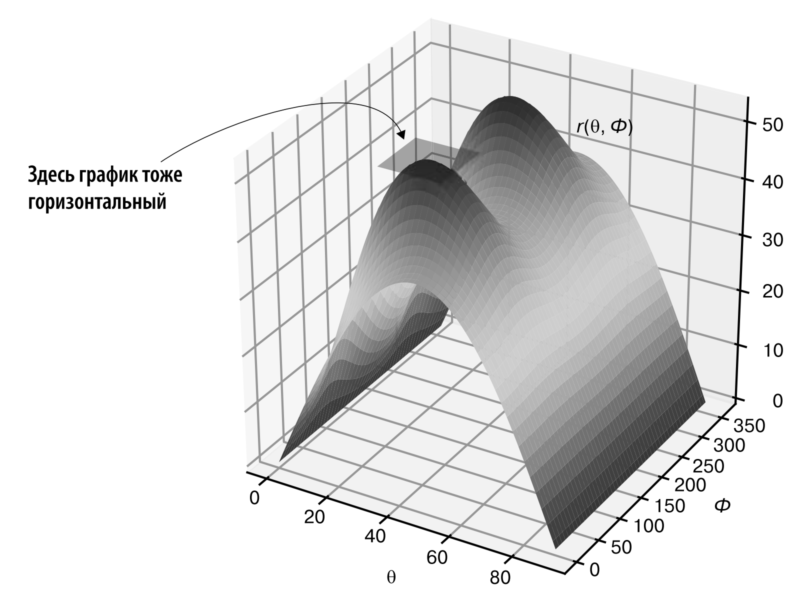


Рис. 12.23. График функции $r(\theta, \phi)$ становится горизонтальным в точках максимума

Прежде чем двинуться дальше, хочу повторить мысль из предыдущего раздела. Точка на графике, где градиент равен нулю, не всегда совпадает с точкой максимального значения. Например, на графике $r(\theta, \phi)$ есть точка между двумя максимумами, где график тоже горизонтальный, а градиент равен нулю (рис. 12.24).

Эта точка не лишена смысла, она указывает наилучший угол θ при стрельбе с поворотом ствола пушки в горизонтальной плоскости на угол $\phi = 180°$, который оказывается наихудшим направлением, потому что это самое крутое направление в гору. Точка, где функция одновременно достигает максимума по одной переменной и минимума по другой, называется *седловой точкой*. Название обусловлено тем, что график функции выглядит как седло.

И снова я не буду вдаваться в подробности идентификации максимумов, минимумов, седловых точек или других мест, где градиент равен нулю, но имейте в виду: чем больше размерностей, тем более странными обстоятельствами может быть обусловлена горизонтальность графика.

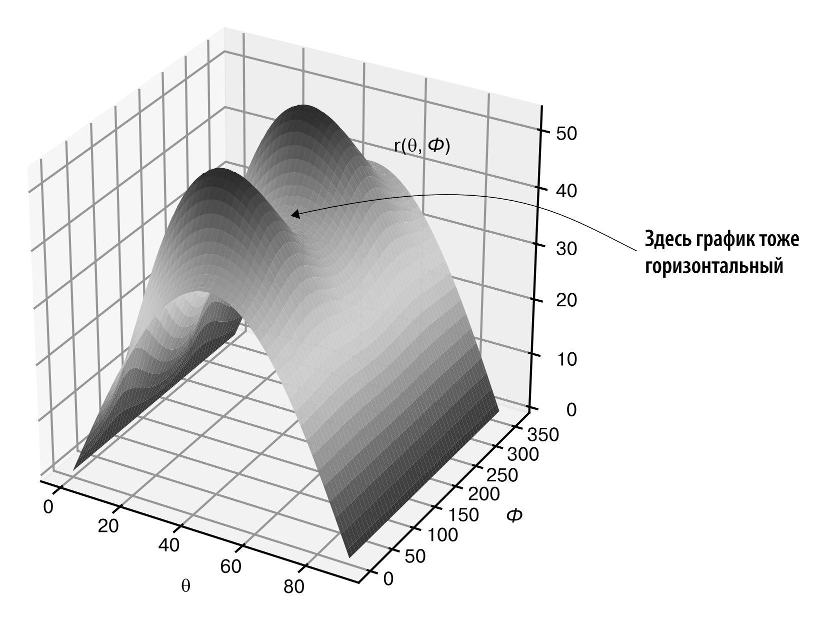


Рис. 12.24. В точке (θ, ϕ) график *r*(θ, ϕ) горизонтальный. Градиент в этой точке равен нулю, но функция не достигает в ней максимального значения

12.4.3. Поиск направления подъема в гору с помощью градиента

Вместо символического решения частных производных сложной функции $r(\theta, \phi)$ можно найти их приблизительные значения. Направление градиента, которое они дают, говорит нам, в каком направлении функция увеличивается быстрее всего в любой произвольной точке. Если двигаться в этом направлении, то это будет движение вверх к максимальному значению. Такая процедура называется *градиентным восхождением*, и мы реализуем ее на Python.

Первый шаг — получить возможность аппроксимации градиента в любой точке. Для этого используем подход, представленный в главе 9, с взятием наклонов малых секущих. Приведу пару соответствующих функций для напоминания:

```
def secant_slope(f,xmin,xmax):
    return (f(xmax) - f(xmin)) / (xmax - xmin)
```

Вычисляет наклон секущей f(x) между значениями x_{min} и x_{max}

```
def approx_derivative(f,x,dx=1e-6):
    return secant_slope(f,x-dx,x+dx)
```

Аппроксимация производной — это секущая между $x - 10^{-6}$ и $x + 10^{-6}$

Чтобы найти аппроксимацию частной производной функции $f(x, y)$ в точке (x_0, y_0), нужно зафиксировать $x = x_0$ и взять производную по y или зафиксировать

$y = y_0$ и взять производную по x. Иначе говоря, частная производная $\partial f/\partial x$ в точке (x_0, y_0) является обыкновенной производной $f(x, y_0)$ по x в точке $x = x_0$. Точно так же частная производная $\partial f/\partial y$ — обычная производная $f(x_0, y)$ по y при $y = y_0$. Градиент — это вектор (кортеж) частных производных:

```
def approx_gradient(f,x0,y0,dx=1e-6):
    partial_x = approx_derivative(lambda x: f(x,y0), x0, dx=dx)
    partial_y = approx_derivative(lambda y: f(x0,y), y0, dx=dx)
    return (partial_x,partial_y)
```

Функция $r(\theta, \phi)$ реализована как функция `landing_distance`, и мы можем использовать в ней специальную функцию `approx_gradient`, представляющую ее градиент:

```
def landing_distance_gradient(theta,phi):
    return approx_gradient(landing_distance_gradient, theta, phi)
```

Она, как и все градиенты, определяет векторное поле, в котором каждой точке пространства назначается определенный вектор. В этом случае она сообщает вектор наибольшего увеличения r в любой точке (θ, ϕ). На рис. 12.25 показан график `landing_distance_gradient`, наложенный на тепловую карту $r(\theta, \phi)$.

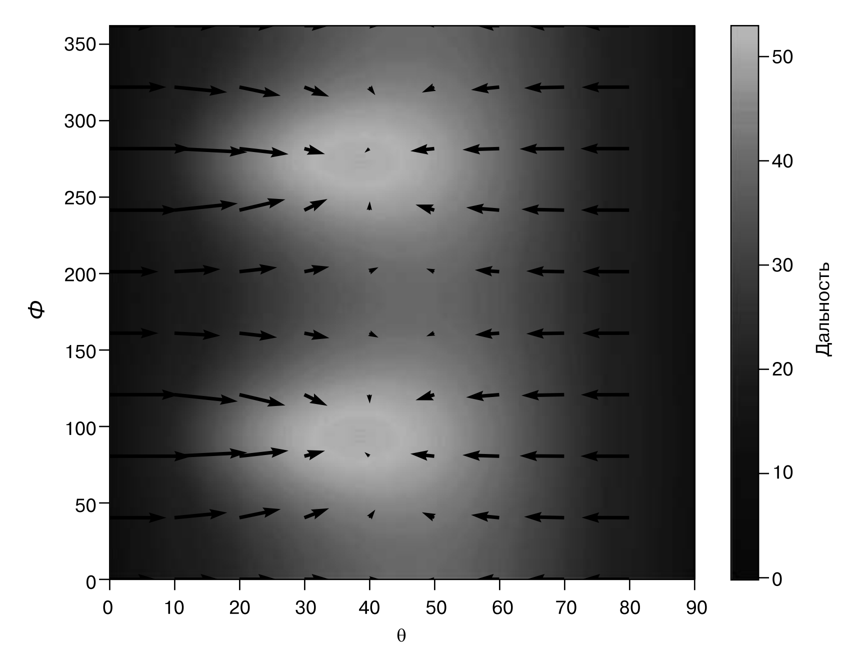


Рис. 12.25. График векторного поля градиента $\nabla r(\theta, \phi)$ поверх тепловой карты функции $r(\theta, \phi)$. Стрелки указывают в направлении увеличения r — к более ярким точкам на тепловой карте

Если увеличить масштаб, то станет ясно видно, что стрелки градиента сходятся в точках максимума функции (рис. 12.26).

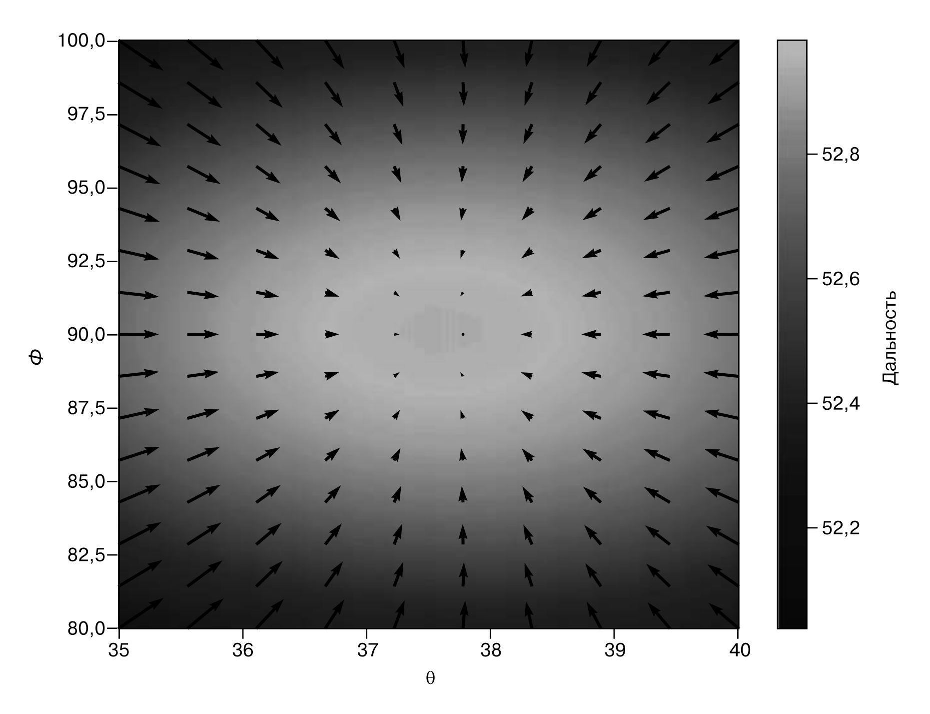


Рис. 12.26. Тот же график, что и на рис. 12.25, в окрестностях точки (θ, ϕ) = (37,5°, 90°) — примерно одного из максимумов

Следующий шаг — реализация алгоритма *градиентного восхождения*: вычисления начинаются с произвольно выбранной точки (θ, ϕ) и движутся за полем градиента, пока не будет достигнут максимум.

12.4.4. Реализация градиентного восхождения

Алгоритм градиентного восхождения принимает функцию, которую нужно максимизировать, а также начальную точку. Наша простая реализация вычисляет градиент в начальной точке и прибавляет его к начальной точке, давая новую точку на некотором расстоянии от исходной в направлении градиента. Повторяя этот процесс, можно двигаться от точки к точке, постепенно приближаясь к максимальному значению.

В конце концов, приблизившись к максимуму, градиент станет близким к нулю и график достигнет плато. Когда градиент близок к нулю, мы больше не сможем идти в гору и алгоритм должен завершиться. Для этого можно задать *допуск*, представляющий наименьшее значение градиента, когда мы еще можем двигаться вперед. Если градиент окажется меньше допуска, можно быть уверенным в том, что график близок к горизонтальной ориентации и мы достигли максимума функции. Вот реализация:

```
def gradient_ascent(f,xstart,ystart,tolerance=1e-6):
    x = xstart
    y = ystart
    grad = approx_gradient(f,x,y)
    while length(grad) > tolerance:
        x += grad[0]
        y += grad[1]
        grad = approx_gradient(f,x,y)
    return x,y
```

Запомнить начальные значения (x, y)

Получить направление наискорейшего подъема из текущей точки (x, y)

Выполнить шаг к новой точке, только если величина градиента больше минимального допуска

Получить градиент в этой новой точке

Вычислить новые значения (x, y) как (x, y) + ∇f(x, y)

По достижении вершины горы вернуть значения x и y

Протестируем эту реализацию, начав со значения $(\theta, \phi) = (36^\circ, 83^\circ)$, которое кажется довольно близким к максимальному:

```
>>> gradient_ascent(landing_distance,36,83)
(37.58114751557887, 89.99992616039857)
```

Результат получился многообещающим! На тепловой карте (рис. 12.27) видно движение от начальной точки $(\theta, \phi) = (36^\circ, 83^\circ)$ к новому местоположению — примерно $(\theta, \phi) = (37{,}58, 90{,}00)$, которое, похоже, находится в самой яркой области.

Чтобы лучше понять, как работает алгоритм, можно проследить траекторию градиентного восхождения через плоскость $\theta\phi$ подобно тому, как мы следили за значениями времени и местоположения в ходе выполнения итераций методом Эйлера:

```
def gradient_ascent_points(f,xstart,ystart,tolerance=1e-6):
    x = xstart
    y = ystart
    xs, ys = [x], [y]
    grad = approx_gradient(f,x,y)
    while length(grad) > tolerance:
        x += grad[0]
        y += grad[1]
        grad = approx_gradient(f,x,y)
        xs.append(x)
        ys.append(y)
    return xs, ys
```

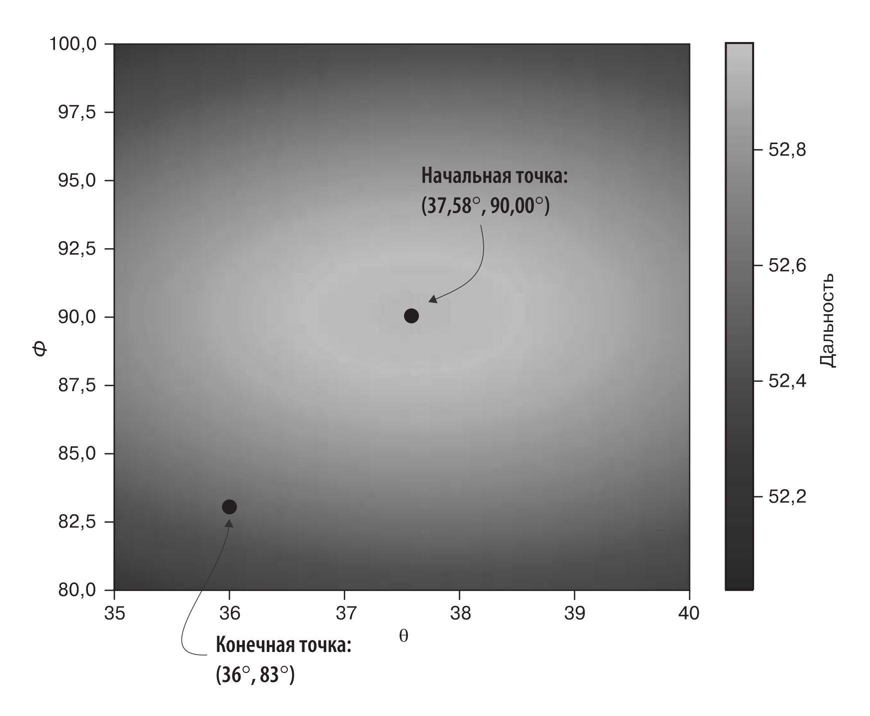


Рис. 12.27. Начальная и конечная точки восхождения по градиенту

Запустив реализацию

```
gradient_ascent_points(landing_distance,36,83)
```

получим два списка значений θ и ϕ, зафиксированных на каждом шаге восхождения. В обоих списках по 855 значений, то есть для завершения градиентного восхождения потребовалось выполнить 855 шагов. Нанеся точки θ и ϕ на тепловую карту (рис. 12.28), можно увидеть путь, которым алгоритм поднимался по графику функции.

Обратите внимание на то, что из-за существования двух максимальных значений путь и конечная точка зависят от выбора начальной точки. Если начать с точки, близкой к $\phi = 90°$, то, скорее всего, будет достигнут этот максимум, но если начать с точки, близкой $\phi = 270°$, алгоритм найдет другой максимум (рис. 12.29).

Углы выстрела (37,58°, 90°) и (37,58°, 270°) максимизируют функцию $r(\theta, \phi)$ и, следовательно, являются углами, обеспечивающими наибольшую дальность полета ядра. Она составляет почти 53 м:

```
>>> landing_distance(37.58114751557887, 89.99992616039857)
52.98310689354378
```

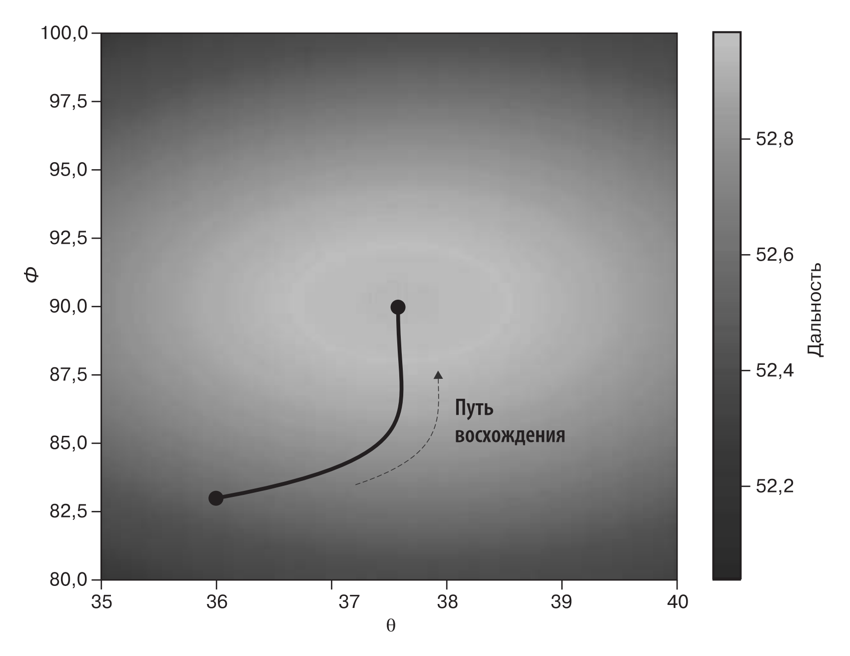


Рис. 12.28. Путь, которым алгоритм градиентного восхождения поднимался до максимального значения функции дальности стрельбы

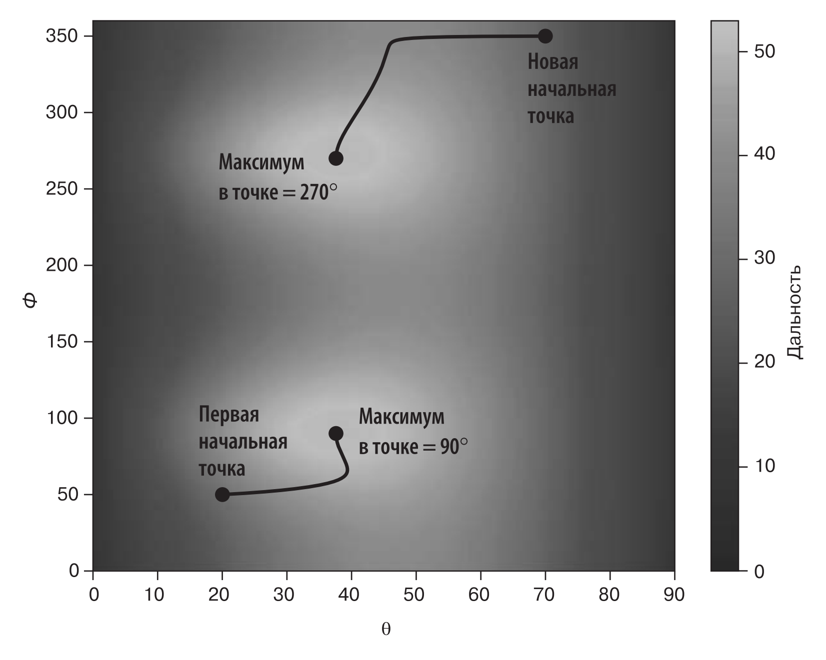


Рис. 12.29. Начиная поиск максимума с разных точек, алгоритм градиентного восхождения может найти разные максимальные значения

Мы можем построить соответствующие траектории полета ядра, как показано на рис. 12.30.

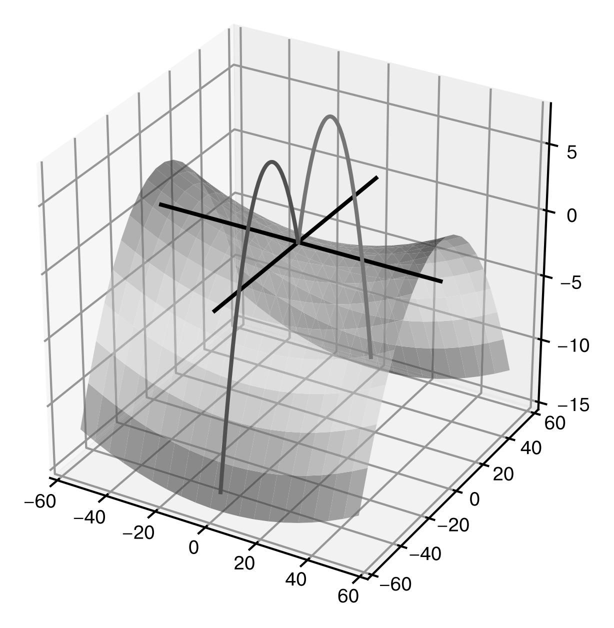


Рис. 12.30. Траектории полета ядра при стрельбе в направлениях, в которых достигается максимальная дальность

По мере знакомства с машинным обучением мы продолжим использовать градиент для оптимизации функций. В частности, будем применять аналог градиентного восхождения, называемый *градиентным спуском*, который находит минимальные значения функций, исследуя пространство параметров в направлении, *противоположном* градиенту, и, соответственно, двигаясь вниз, а не вверх. Поскольку градиентное восхождение и спуск могут выполняться автоматически, с их помощью машины могут самостоятельно искать оптимальные решения задач.

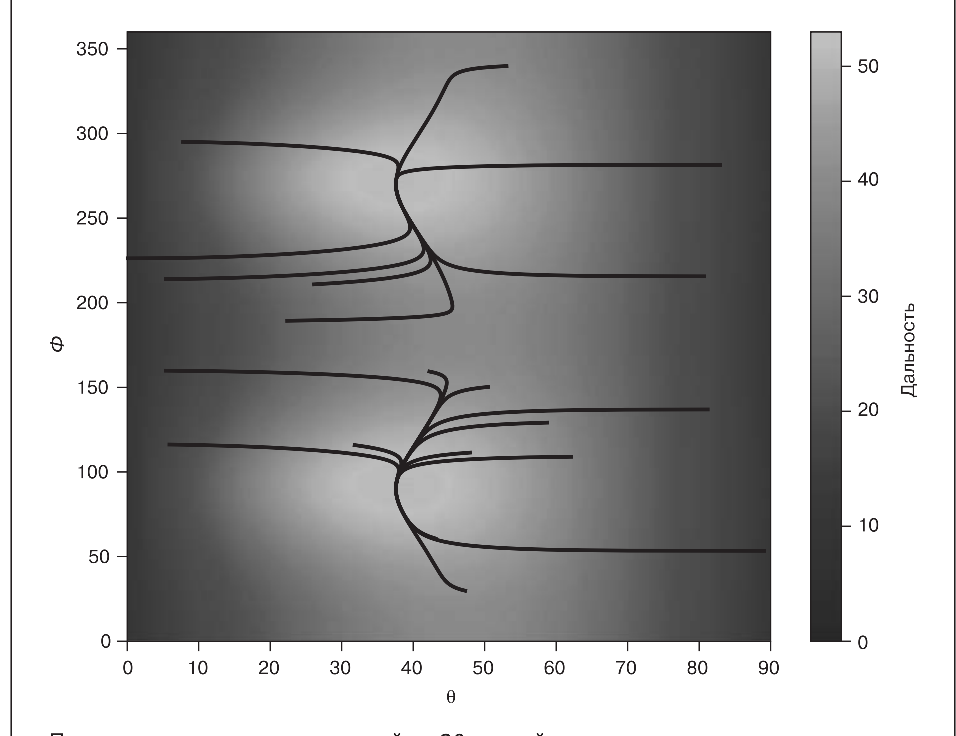
12.4.5. Упражнения

Упражнение 12.13. Нарисуйте на тепловой карте пути градиентного восхождения из 20 случайно выбранных точек. Все пути должны заканчиваться на одной из двух точек максимума.

Решение. Имея построенную тепловую карту, можно выполнить следующий код, чтобы совершить 20 случайных восхождений по градиенту и нарисовать их пути:

```
from random import uniform
for x in range(0,20):
    gap = gradient_ascent_points(landing_distance,
                                 uniform(0,90),
                                 uniform(0,360))
plt.plot(*gap,c='k')
```

Результат показывает, что все пути ведут в одни и те же две точки.



Пути градиентных восхождений из 20 случайных начальных точек

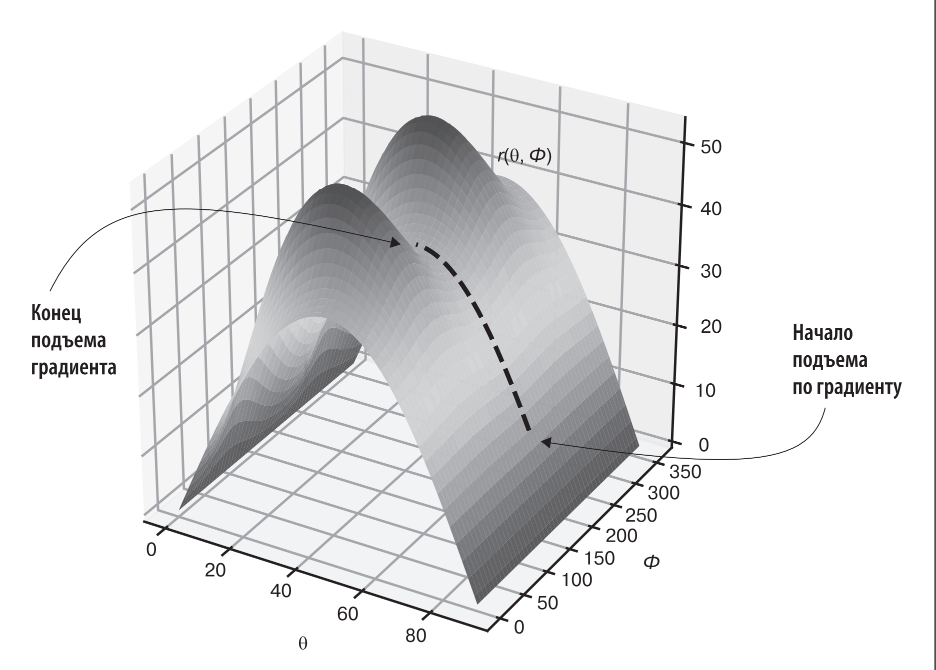
Упражнение 12.14. Мини-проект. Найдите символически частные производные $\partial r/\partial\theta$ и $\partial r/\partial\phi$ и напишите формулу градиента $\nabla r(\theta, \phi)$.

Упражнение 12.15. Найдите точку на графике $r(\theta, \phi)$, где градиент равен нулю, но функция не максимальна.

Решение. Можно обмануть алгоритм градиентного восхождения, начав поиск с точки $\phi = 180°$. Благодаря симметрии рельефа мы видим, что $\partial r/\partial\phi = 0$ везде, где $\phi = 180°$, поэтому алгоритм градиентного восхождения не увидит причины покинуть линию, где $\phi = 0$:

```
>>> gradient_ascent(landing_distance,0,180)
(46.122613357930206, 180.0)
```

Это оптимальный угол стрельбы при фиксированном повороте ствола на угол ϕ, равный 0 или 180°, который является наихудшим углом, потому что стрелять приходится в гору.



Выбрав начальную точку на поперечном сечении, где $\partial r/\partial\phi = 0$, можно обмануть алгоритм градиентного восхождения

Упражнение 12.16. Сколько шагов потребуется алгоритму градиентного восхождения, чтобы достичь начала координат, двигаясь из точки (36, 83)? Вместо того чтобы прыгать на одну величину градиента, попробуйте прыгать на 1,5 величины градиента. Проверьте, действительно ли так можно добраться до цели за меньшее количество шагов. Что произойдет, если на каждом шаге прыгать еще дальше?

Решение. Введем параметр `rate` скорости расчета градиентного восхождения, который определяет, насколько быстрым оно будет. Чем выше скорость, тем больше мы доверяем текущему вычисленному значению градиента и прыгаем в этом направлении:

```
def gradient_ascent_points(f,xstart,ystart,rate=1,tolerance=1e-6):
    ...
    while length(grad) > tolerance:
        x += rate * grad[0]
        y += rate * grad[1]
        ...
    return xs, ys
```

Вот функция, подсчитывающая количество шагов, необходимых для сходимости процесса градиентного восхождения:

```
def count_ascent_steps(f,x,y,rate=1):
    gap = gradient_ascent_points(f,x,y,rate=rate)
    print(gap[0][-1],gap[1][-1])
    return len(gap[0])
```

Первоначальной версии градиентного подъема с параметром `rate`, равным 1, требуется 855 шагов:

```
>>> count_ascent_steps(landing_distance,36,83)
855
```

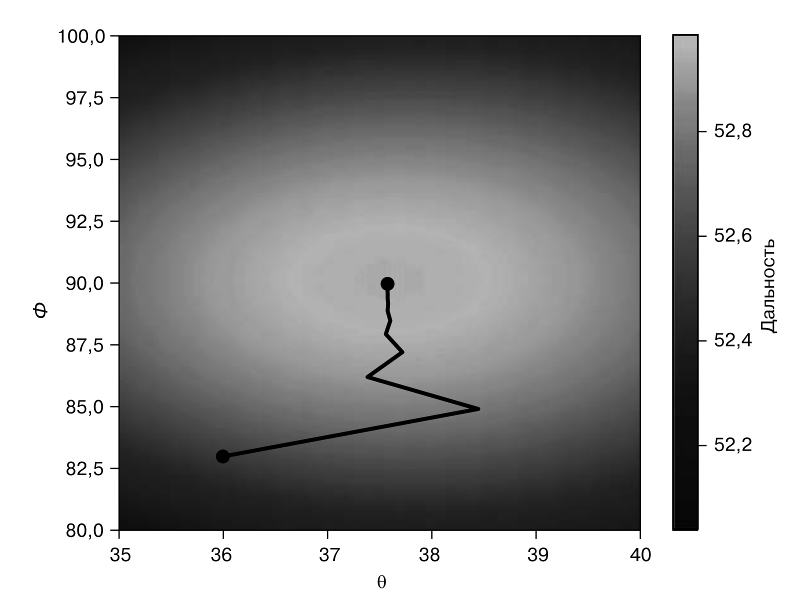
С параметром `rate = 1.5` на каждом шаге происходит прыжок на полторы величины градиента. Неудивительно, что в этом случае алгоритм добирается до максимума быстрее — всего за 568 шагов:

```
>>> count_ascent_steps(landing_distance,36,83,rate=1.5)
568
```

Попробовав еще несколько значений, можно заметить, что увеличение скорости (`rate`) приводит к решению за еще меньшее количество шагов:

```
>>> count_ascent_steps(landing_distance,36,83,rate=3)
282
>>> count_ascent_steps(landing_distance,36,83,rate=10)
81
>>> count_ascent_steps(landing_distance,36,83,rate=20)
38
```

Однако не следует слишком жадничать! Используя коэффициент 20, мы получаем ответ за меньшее количество шагов, но некоторые из них, похоже, перепрыгивают ответ, и на следующем шаге алгоритму приходится возвращаться. Если задать слишком высокую скорость, алгоритм может даже отдаляться от решения. В этом случае говорят, что он расходится, а не сходится.

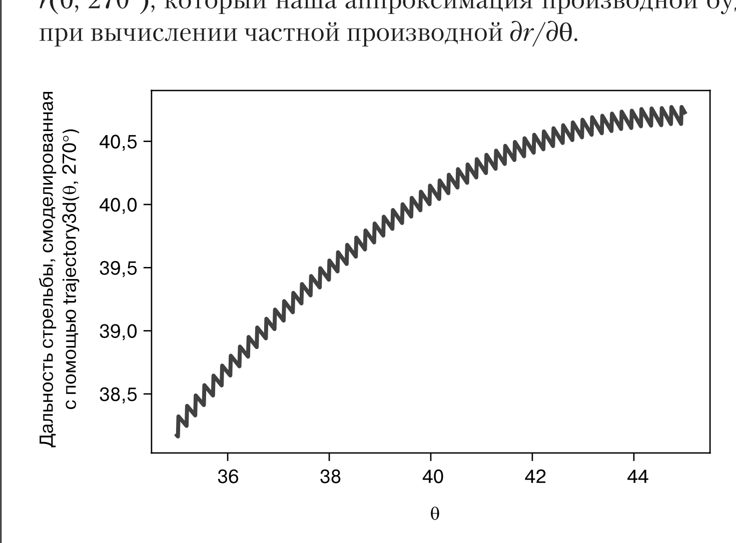


Градиентное восхождение со скоростью 20. Алгоритм изначально превышает максимальное значение θ, и ему приходится возвращаться назад

Если задать скорость равной 40, то алгоритм градиентного восхождения не сойдется. Каждый прыжок будет промахиваться дальше предыдущего, и исследование пространства параметров убежит в бесконечность.

Упражнение 12.17. Что случится, если попытаться вызвать функцию `gradient_ascent` напрямую, используя смоделированные результаты для r как функцию θ и ϕ вместо вычисленных результатов?

Решение. Получится некрасивый результат. Причина в том, что смоделированные результаты зависят от численных оценок (например, от определения момента падения ядра на землю), поэтому они колеблются при небольших изменениях углов выстрела. Вот график поперечного сечения $r(\theta, 270°)$, который наша аппроксимация производной будет учитывать при вычислении частной производной $\partial r/\partial\theta$.



Сечение смоделированных траекторий показывает, что модель не дает гладкой функции $r(\theta, \phi)$

Значение производной сильно колеблется, поэтому алгоритм градиентного восхождения перемещается в случайных направлениях.

КРАТКИЕ ИТОГИ ГЛАВЫ

- Траекторию движения объекта можно смоделировать с помощью метода Эйлера, фиксируя все моменты времени и координаты местоположения во время движения. Есть возможность вычислить такие факты о траектории, как координаты конечной точки или прошедшее время.
- Изменение параметра модели, такого как угол выстрела из пушки, может привести к изменению результата, например, дать другую дальность полета

пушечного ядра. Чтобы найти угол, дающий максимальную дальность, полезно выразить дальность как функцию угла $r(\theta)$.

- Максимальные значения гладкой функции $f(x)$ находятся там, где производная $f'(x)$ равна нулю. Однако следует быть осторожными, потому что не всегда $f'(x) = 0$ соответствует максимальному значению функции f, — нулевой производной может соответствовать также минимальное значение или точка, в которой функция f временно перестала изменяться.
- Чтобы оптимизировать функцию двух переменных, такую как функция дальности стрельбы r от вертикального угла θ и горизонтального угла ϕ поворота ствола пушки, необходимо исследовать двухмерное пространство всех возможных входных данных (θ, ϕ) и выяснить, какая пара дает оптимальное значение.
- Максимальное и минимальное значения гладкой функции двух переменных $f(x, y)$ находятся в точках, где обе частные производные равны нулю, то есть $\partial f/\partial x = 0$ и $\partial f/\partial y = 0$, поэтому выполняется также условие $\nabla f(x, y) = 0$ (по определению). Частные производные могут быть равны нулю и в седловой точке, где функция имеет минимальное значение по одной переменной и максимальное — по другой.
- Алгоритм градиентного восхождения находит приблизительное максимальное значение функции $f(x, y)$, начиная с произвольно выбранной точки в двухмерном пространстве и двигаясь в направлении градиента $\nabla f(x, y)$. Поскольку градиент указывает в направлении скорейшего увеличения функции f, этот алгоритм находит точки (x, y) с возрастающими значениями f. Алгоритм завершается, когда градиент приближается к нулю.

13

Анализ звуковых волн с использованием рядов Фурье

В этой главе

- Определение и воспроизведение звуковых волн с помощью Python и PyGame.
- Преобразование синусоидальных функций в музыкальные ноты.
- Объединение двух звуков сложением их функций.
- Разложение функции звуковой волны в ряд Фурье для получения составляющих ее музыкальных нот.

В части II почти все свое внимание мы уделяли использованию матанализа для моделирования движущихся объектов. В этой главе я покажу совершенно другое приложение — предназначенное для обработки аудиоданных. Цифровые аудиоданные — это компьютерное представление *звуковых волн*, вызывающих повторяющиеся изменения давления воздуха, которые наши уши воспринимают как звук. Мы будем рассматривать звуковые волны как функции, которые можно складывать и масштабировать как векторы и к которым можно применять дифференциальное исчисление, чтобы выяснить, какие ноты они представляют. В исследовании звуковых волн мы используем многое из того, что узнали о линейной алгебре и математическом анализе в предыдущих главах.

Я не буду слишком углубляться в физику звуковых волн, но вам будет полезно знать базовые принципы. То, что мы воспринимаем как звук, — это не само

давление воздуха, а скорее быстрые изменения давления, которые вызывают вибрацию наших барабанных перепонок. Например, играя на скрипке, музыкант проводит смычком по струне и заставляет ее вибрировать. Вибрирующая струна вызывает колебания давления окружающего ее воздуха, которые распространяются по нему в виде звуковых волн, пока не достигнут уха. В этот момент барабанные перепонки начинают вибрировать с той же частотой, и мы воспринимаем эти вибрации как звук (рис. 13.1).

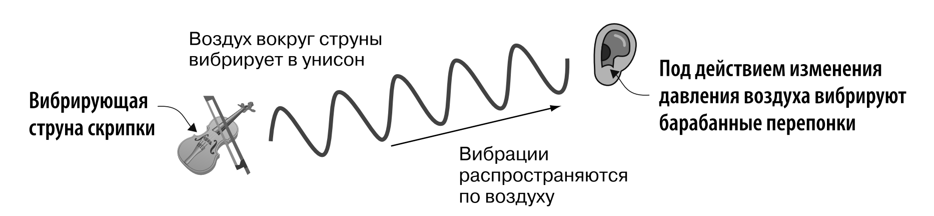


Рис. 13.1. Упрощенная схема извлечения из скрипки звука, достигающего барабанной перепонки

Цифровой аудиофайл можно рассматривать как функцию, описывающую вибрацию во времени. Программное обеспечение для воспроизведения аудиофайлов интерпретирует функцию и посылает сигналы динамикам, заставляя их вибрировать соответствующим образом и создавать звуковые волны аналогичной формы в окружающем воздухе. Для наших целей не имеет значения, что именно представляет функция, и мы можем свободно интерпретировать ее как описание изменения давления воздуха во времени (рис. 13.2).

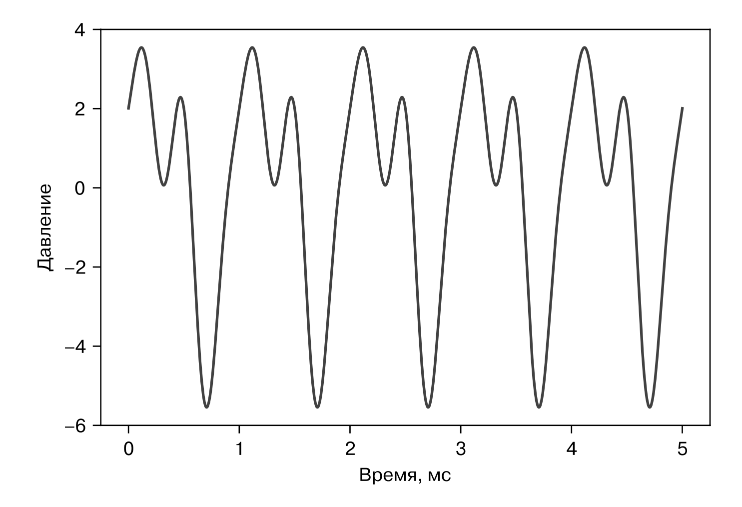


Рис. 13.2. Представление звуковых волн как функции, описывающей изменение давления воздуха во времени

Привлекательные звуки, такие как музыкальные ноты, порождаются звуковыми волнами с повторяющимся рисунком, как показано на рис. 13.2. Скорость повторения функции называется *частотой* и определяет высоту звучания музыкальной ноты. Качество, или *тембр*, звука зависит от формы повторяющегося узора. Например, от него зависит, на что будет больше похоже звучание — на скрипку, трубу или человеческий голос.

13.1. ОБЪЕДИНЕНИЕ ЗВУКОВЫХ ВОЛН И ИХ РАЗЛОЖЕНИЕ

На протяжении всей этой главы мы будем выполнять математические операции над функциями и использовать Python для их воспроизведения в виде звуков. Основное внимание уделим двум операциям — объединению имеющихся звуковых волн для создания новых и разложению сложных звуковых волн на более простые. Например, мы попробуем объединить несколько музыкальных нот в аккорд, а затем разложить его на составляющие музыкальные ноты.

Однако, прежде чем приступить к экспериментам, нужно познакомиться с основными строительными блоками — звуковыми волнами и музыкальными нотами. Для начала я покажу вам, как с помощью Python превратить последовательность чисел, представляющую звуковую волну, в звук, исходящий из динамиков. Чтобы создать звук, соответствующий заданной функции, нужно извлечь некоторые значения y из графика функции и передать их аудиобиблиотеке в виде массива. Этот процесс называется *выборкой* (рис. 13.3).

В основном в роли функций звуковых волн мы будем использовать *периодические функции*, графики которых имеют одну и ту же повторяющуюся форму. В частности, задействуем *синусоидальные функции* — семейство периодических функций, включающее синус и косинус, которые воспроизводят естественно звучащие музыкальные ноты. После выборки из них последовательности чисел создадим функции на Python для воспроизведения музыкальных нот.

Научившись воспроизводить отдельные ноты, мы напишем код, объединяющий разные ноты в аккорды и другие сложные звуки. Такое объединение будет производиться путем сложения функций, определяющих отдельные звуковые волны. Как вы увидите сами, объединение нескольких музыкальных нот может составить аккорд, а объединение десятков нот может дать довольно интересные и качественно разные звуки.

Нашей последней целью будет разложение функции, представляющей произвольную звуковую волну, на сумму чистых музыкальных нот и соответствующих

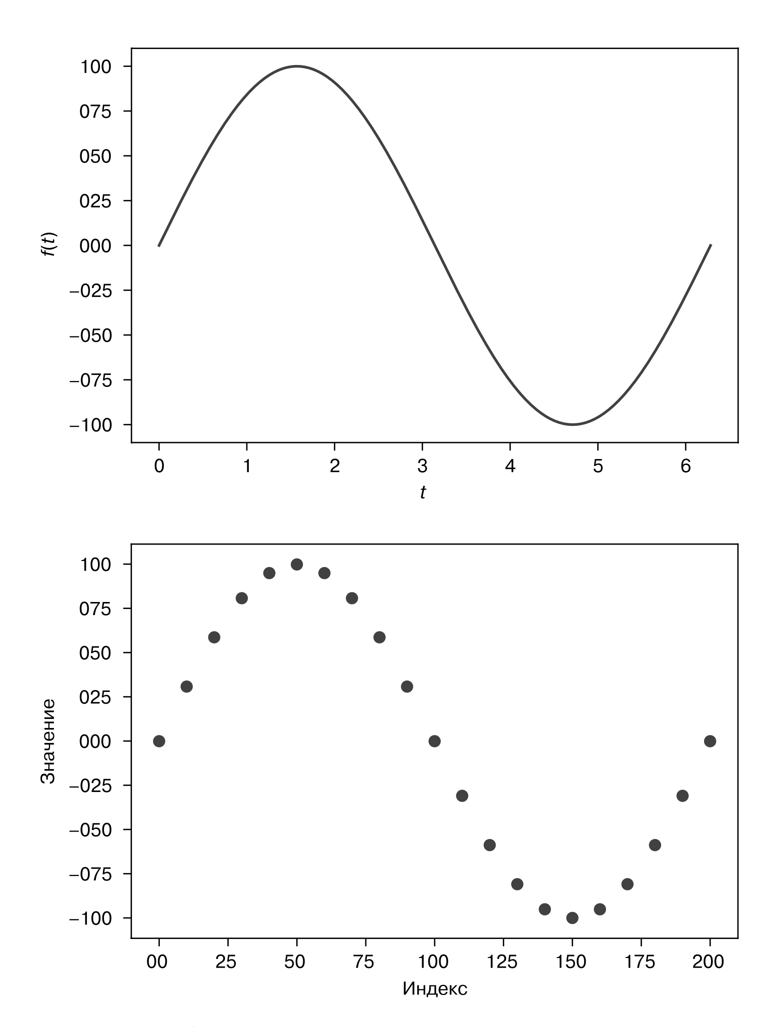


Рис. 13.3. Выборка некоторых значений *y (внизу)* из графика функции *f*(*t*) *(вверху)* для передачи аудиобиблиотеке

им громкостей (рис. 13.4). Такое разложение называется *рядом Фурье*. Получив звуковые волны, составляющие ряд Фурье, мы сможем воспроизвести их вместе и получить исходный звук.

Математически поиск рядов Фурье означает запись функции в виде суммы, или, точнее, линейной комбинации функций синуса и косинуса. Эта процедура и ее варианты — одни из самых важных алгоритмов всех времен. Методы, подобные тем, которые мы рассмотрим, используются в обычных приложениях, например, для сжатия MP3, а также в более грандиозных работах, таких как обнаружение гравитационных волн, недавно удостоенной Нобелевской премии.

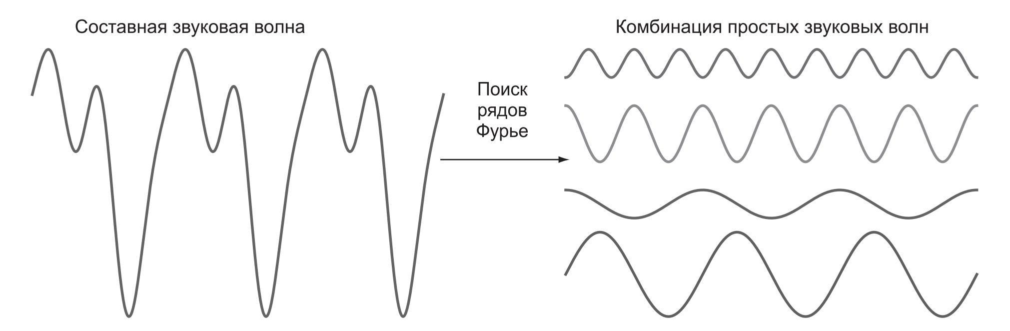


Рис. 13.4. Разложение функции звуковой волны в комбинацию более простых функций с помощью ряда Фурье

Но одно дело смотреть на графики звуковых волн и совсем другое — слышать, как они исходят из динамиков. Так давайте же создадим шум!

13.2. ВОСПРОИЗВЕДЕНИЕ ЗВУКОВЫХ ВОЛН В PYTHON

Чтобы воспроизвести звуки в Python, обратимся к библиотеке PyGame, которую мы использовали в нескольких предыдущих главах. В частности, задействуем функцию, которая принимает массив чисел и воспроизводит звук. В качестве первого шага возьмем случайную последовательность чисел и на ее основе с помощью PyGame напишем код для интерпретации и воспроизведения звука. Это будет всего лишь *шум* (да, это технический термин!), а не красивая музыка, но нам нужно с чего-то начать.

Создав некоторый шум, попробуем воспроизвести немного более привлекательный звук, запустив тот же процесс с упорядоченной последовательностью чисел, включающей повторяющиеся фрагменты. Так мы подготовимся к следующему разделу, где получим последовательность повторяющихся чисел путем выборки из периодической функции.

13.2.1. Воспроизведение первого звука

Прежде чем передать библиотеке PyGame массив чисел, представляющих звук, нужно сообщить ей, как интерпретировать числа. Это требует знания некоторых технических подробностей об аудиоданных, и я объясню их, чтобы вы знали, как PyGame их интерпретирует, но эти детали не будут иметь решающего значения для остальной части главы.

В этом приложении мы используем соглашения, обычно применяемые при производстве аудио-компакт-дисков. В частности, 1 с звучания представим массивом из 44 100 значений, каждое из которых является 16-битным целым числом (от –32 768 до 32 767) и задает интенсивность звука на каждом из 44 100 шагов в секунду. Этот способ незначительно отличается от способа, примененного для представления изображений в главе 6. Вместо массива значений, определяющих яркость пикселов, у нас будет массив значений, определяющих интенсивность звуковой волны в разные моменты времени. В конце концов мы получим эти числа как координаты *y* точек на графике звуковых волн, но пока будем выбирать их случайным образом, чтобы создать некоторый шум.

Мы также задействуем один *канал*, то есть будем воспроизводить только одну звуковую волну, а не две, как в стереозвуке: одну в левом динамике и одну в правом. Кроме того, настроим *битовую глубину* звука. Если частоту можно сравнить с разрешением изображения, то битовая глубина подобна количеству допустимых цветов пикселов: чем больше битовая глубина, тем шире диапазон интенсивности звука. Для представления цвета пикселов мы использовали три числа, каждое в диапазоне от 0 до 256, а здесь возьмем одно 16-битное число для представления интенсивности звука в определенный момент времени. Итак, определившись с основными параметрами, приступим к программному коду. Вначале импортируем пакет PyGame и инициализируем библиотеку воспроизведения звука:

```
>>> import pygame, pygame.sndarray
>>> pygame.mixer.init(frequency=44100,
                      size=-16,
                      channels=1)
```

Значение –16 указывает, что битовая глубина будет равна 16 и на вход будут подаваться 16-битные целые числа со знаком в диапазоне от –32 768 до 32 767

Начнем с самого простого из возможных примеров — сгенерируем 1 с звука, создав массив NumPy из 44 100 случайных целых чисел в диапазоне от –32 768 до 32 767. Сделать это можно одной строкой с помощью функции `randint` из библиотеки NumPy:

```
>>> import numpy as np
>>> arr = np.random.randint(-32768, 32767, size=44100)
>>> arr
array([-16280, 30700, -12229, ..., 2134, 11403, 13338])
```

Чтобы посмотреть, как выглядит график этой звуковой волны, можно нанести на него несколько первых значений. Я включил функцию `plot_sequence` в примеры исходного кода для книги, чтобы помочь вам быстро построить график на основе массива целочисленных значений. Вызов `plot_sequence(arr, max=100)` выведет график с первыми 100 значениями этого массива. По сравнению с числами, взятыми из гладкой функции, они разбросаны по всей плоскости координат (рис. 13.5).

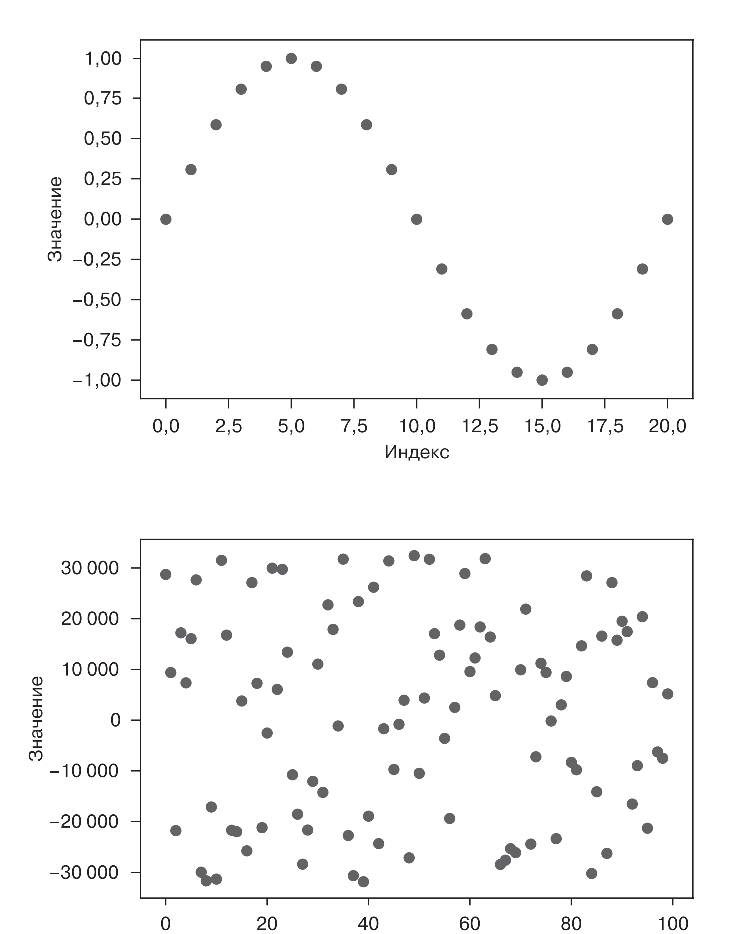


Рис. 13.5. Выборочные значения звуковой волны *(вверху)* и случайные значения *(внизу)*

Если соединить точки отрезками, то можно увидеть нечто напоминающее график функции, соответствующий этому периоду времени. На рис. 13.6 изображены два графика, построенных на основе одного и того же массива чисел. На первом показаны 100 точек и на втором — 441. Эти данные совершенно случайные, так что в графиках нет ничего интересного, но это будет первая звуковая волна, которую мы воспроизведем.

Поскольку 44 100 значений определяют 1 с звучания, 441 значение на втором графике определяют звучание в течение первой сотой доли секунды. Теперь мы можем воспроизвести звук с помощью библиотеки.

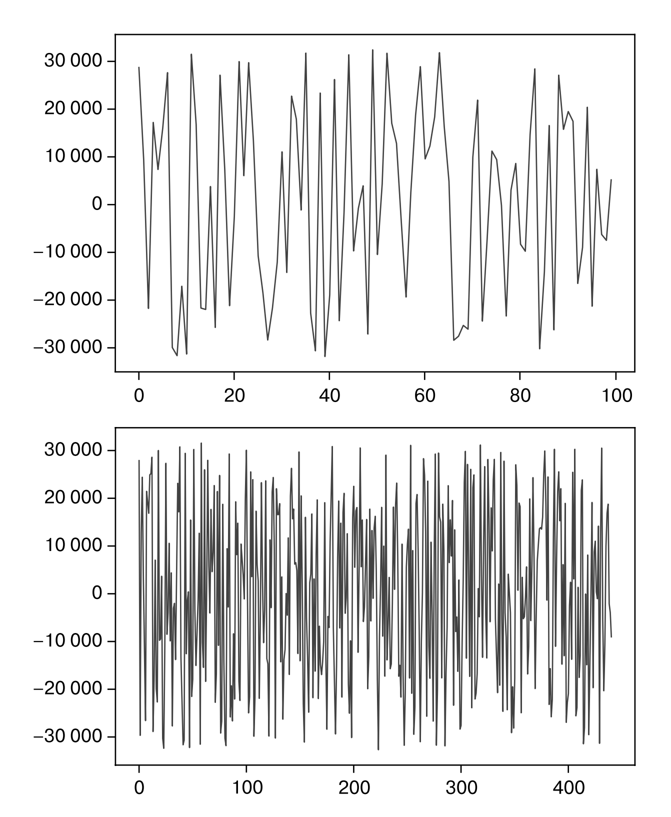


Рис. 13.6. Первые 100 значений *(вверху)* и первые 441 значение *(внизу)* в форме графика функции

ВНИМАНИЕ

Прежде чем запустить следующие несколько строк кода на Python, уменьшите громкость своего динамика. Первый звук, который мы воспроизведем, будет не самым приятным, так что не стоит травмировать уши.

Воспроизвести звук можно вызовом

```
sound = pygame.sndarray.make_sound(arr)
sound.play()
```

В результате вы должны услышать шум, напоминающий разряды статического электричества, как если бы включили радио, не настроив его на радиостанцию. Такая звуковая волна, состоящая из случайных значений, называется *белым шумом*.

Единственное, что можно настроить в белом шуме, — это громкость. Человеческое ухо реагирует на изменения давления, и чем шире диапазон изменения значений звуковой волны, тем сильнее меняется давление и тем более громким воспринимается звук. Если этот белый шум оказался для вас неприятно громким,

то попробуйте создать более тихую версию, сгенерировав звуковые данные из меньших чисел. Например, следующий код генерирует белый шум из чисел в диапазоне от –10 000 до 10 000:

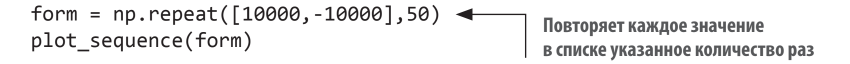
```
arr = np.random.randint(-10000, 10000, size=44100)
sound = pygame.sndarray.make_sound(arr)
sound.play()
```

Этот звук должен быть почти идентичен первому белому шуму, только тише. Громкость звуковой волны зависит от значений функции, и эта мера громкости называется *амплитудой* волны. В данном случае, поскольку значения отклоняются не более чем на 10 000 единиц от среднего значения 0, считается, что амплитуда равна 10 000.

Хотя некоторые люди находят белый шум успокаивающим, он не очень интересен. Воспроизведем более интересный звук, а именно музыкальную ноту.

13.2.2. Воспроизведение музыкальной ноты

Когда мы слышим музыкальную ноту, наши уши улавливают закономерность вибраций, отличающуюся от случайного белого шума. Мы можем составить серию из 44 100 чисел, обладающую очевидной закономерностью, и вы услышите, что они воспроизводят музыкальную ноту. Например, для начала 50 раз повторим число 10 000, затем 50 раз повторим число –10 000. Я выбрал 10 000, так как мы только что убедились, что это достаточно большая амплитуда для воспроизводимого звука. На рис. 13.7 показан график, образуемый первыми 100 числами из последовательности, возвращаемой следующим фрагментом кода:



```
form = np.repeat([10000,-10000],50)
plot_sequence(form)
```

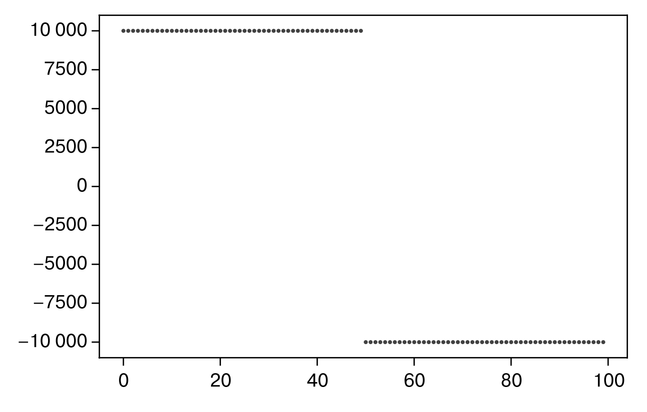



Рис. 13.7. График последовательности, состоящей из числа 10 000, повторенного 50 раз, за которым следует число –10 000, повторенное 50 раз

Повторив эту последовательность из 100 чисел 441 раз, получим 44 100 значений, определяющих 1 с звука. Для этого можно использовать еще одну удобную функцию из библиотеки NumPy — `tile`, которая повторяет заданный массив заданное количество раз:

```
arr = np.tile(form,441)
```

На рис. 13.8 показан график первых 1000 значений массива в виде точек, соединенных отрезками. Как видите, он прыгает вверх-вниз между значениями 10 000 и –10 000 через каждые 50 точек. Это означает, что шаблон повторяется через каждые 100 точек.

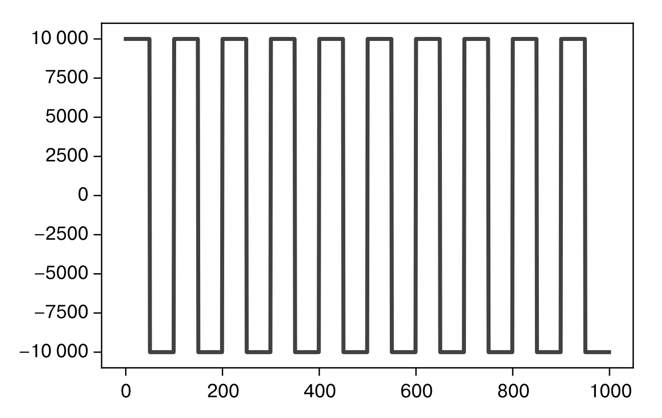


Рис. 13.8. График первой 1000 из 44 100 чисел демонстрирует повторяющуюся закономерность

Графики такого вида называют *прямоугольной волной*, потому что они имеют резкие углы в 90°. (Обратите внимание на то, что вертикальные линии здесь получились только потому, что в последовательности нет значений между 10 000 и –10 000.)

Последовательность из 44 100 чисел представляет 1 с звучания, поэтому 1000 чисел, изображенных на рис. 13.8, соответствуют 1/44,1 с (или 0,023 с). Воспроизведение этих аудиоданных с использованием следующих строк дает четкую музыкальную ноту. Это примерно нота A, или A_4 в научной форме записи высоты тона[1]. Ее можно прослушать, вызвав ту же функцию `play()`, что и в разделе 13.2.1:

```
sound = pygame.sndarray.make_sound(arr)
sound.play()
```

Количество повторений в секунду (в данном случае 441 повторение) называется *частотой* звуковой волны и определяет высоту ноты. Частота повторения измеряется в *герцах* [Гц], где 441 Гц означает то же самое, что 441 раз в секунду.

[1] A — это нота ля, A4 — ля первой октавы. — *Примеч. пер.*

Нота a (ля) создается звуковой волной с частотой 440 Гц, однако частота 441 Гц довольно близка к этой ноте и удобно делит частоту дискретизации в 44 100 значений в секунду, принятую при производстве аудио-компакт-дисков.

Интересные звуковые волны порождаются периодическими функциями, состоящими из закономерностей, которые повторяются через фиксированные интервалы, подобно прямоугольной волне на рис. 13.8. Повторяющаяся закономерность, образующая прямоугольную волну, состоит из 100 чисел, она повторяется 441 раз, чтобы получить 44 100 чисел для 1 с звучания. Это соответствует частоте повторений 441 Гц или одному повторению каждые 0,0023 с. То, что наше ухо воспринимает как музыкальную ноту, определяется частотой повторения. В следующем разделе мы воспроизведем звуки, соответствующие наиболее важным периодическим функциям, синусу и косинусу, с разными частотами.

13.2.3. Упражнения

Упражнение 13.1. Музыкальная нота a (ля) была сформирована прямоугольной волной с частотой 441 Гц. Создайте аналогичную волну с частотой 350 Гц, чтобы получить музыкальную ноту F (фа).

Решение. К счастью, частота 44 100 Гц делится на 350 без остатка: 44 100/350 = 126. Объединив в последовательность 63 значения 10 000 и 63 значения –10 000, можно повторить эту последовательность 350 раз, чтобы получить звук, длящийся 1 с. Получившаяся нота звучит ниже ноты A (ля) и действительно является нотой F (фа):

```
form = np.repeat([10000,-10000],63)
arr = np.tile(form,350)
sound = pygame.sndarray.make_sound(arr)
sound.play()
```

13.3. ПРЕОБРАЗОВАНИЕ СИНУСОИДАЛЬНОЙ ВОЛНЫ В ЗВУК

Звук, который мы получили воспроизведением прямоугольной волны, был узнаваемой музыкальной нотой, но с не очень естественным звучанием, потому что в природе вибрации имеют форму, отличную от прямоугольной. Чаще встречаются *синусоидальные* вибрации, то есть график такой природной вибрации будет похож на график синуса или косинуса. Кроме того, эти функции более естественны с математической точки зрения, поэтому их можно использовать в качестве

строительных блоков гармоничных звуков, которые мы собираемся создавать. Создав выборку с последовательностью значений и передав ее в PyGame, вы сможете услышать разницу между прямоугольной и синусоидальной волной.

13.3.1. Создание звука на основе синусоидальных функций

Синусоидальные функции (синус и косинус), которые мы уже не раз применяли в этой книге, по своей сути являются периодическими функциями. Их входные аргументы интерпретируются как углы: сделав полный оборот на 360°, или 2π рад, вы будет смотреть в том же направлении, что и перед поворотом, так и функции синуса и косинуса снова и снова возвращают те же значения. Следовательно, значения $\sin(t)$ и $\cos(t)$ повторяются с каждым увеличением аргумента на 2π единиц, как показано на рис. 13.9.

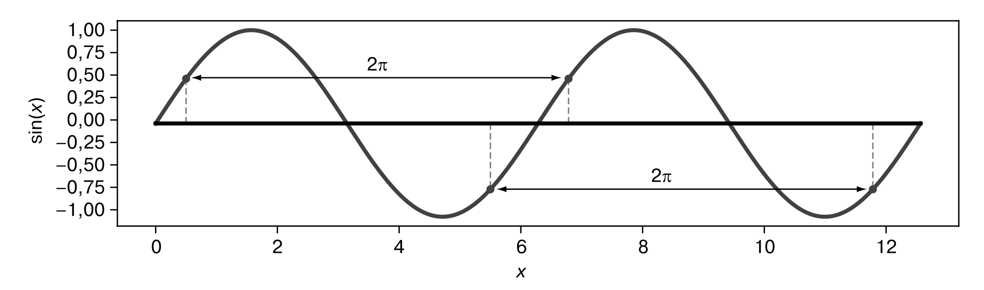


Рис. 13.9. С каждым увеличением аргумента на 2π единиц функция sin (*t*) возвращает одно и то же значение

Интервал повторения называется *периодом* периодической функции, то есть для синуса и косинуса период равен 2π. Нарисовав их графики (рис. 13.10), вы увидите, что они снова и снова повторяют одну и ту же последовательность значений на интервалах от 0 до 2π, от 2π до 4π, от 4π до 6π и т. д.

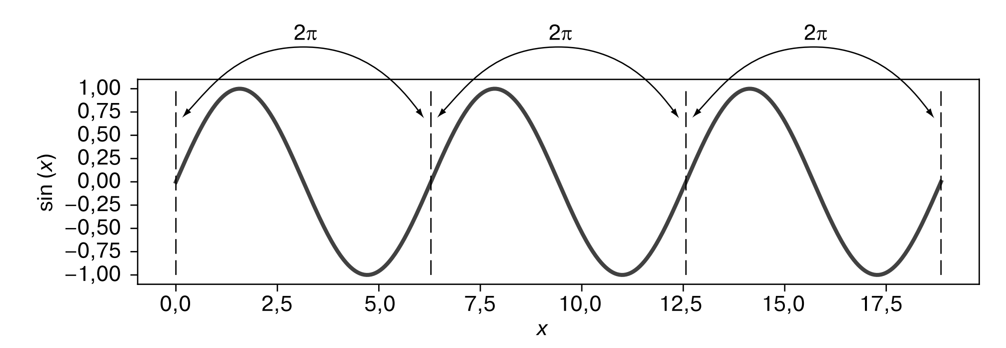


Рис. 13.10. Поскольку функция синуса периодическая с периодом 2π, ее график повторяется на каждом интервале 2π

Единственное отличие функции синуса от функции косинуса в том, что график последней смещен на $\pi/2$ единицы влево, но точно так же повторяется через каждые 2π единицы (рис. 13.11).

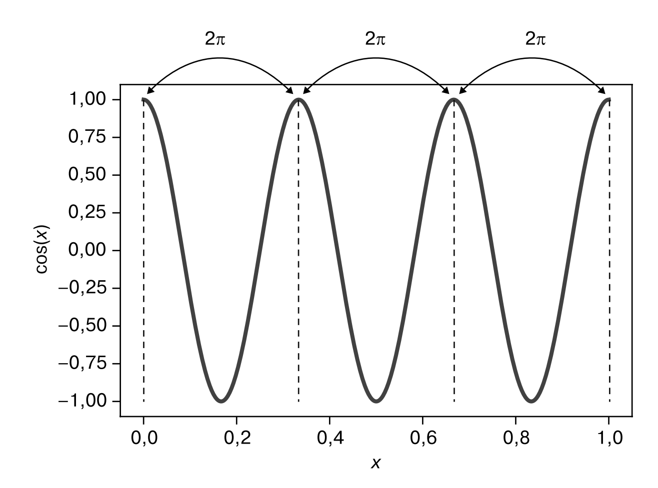


Рис. 13.11. График функции косинуса имеет ту же форму, что и график функции синуса, но смещен влево. Он тоже повторяется через каждые 2π единиц

Одно повторение каждые 2π секунд соответствует частоте $1/2\pi$, или около 0,159 Гц, — слишком маленькой, чтобы этот звук мог услышать человек. Амплитуда 1,0 тоже слишком маленькая для 16-битного аудиосигнала. Чтобы исправить эту проблему, напишем функцию `make_sinusoid(frequency,amplitude)`, которая создаст синусоидальную функцию, растянутую или сжатую по вертикали и горизонтали, чтобы получить желаемую частоту и амплитуду. Частота 441 Гц и амплитуда 10 000 должны давать хорошо слышимую звуковую волну.

Написав эту функцию, мы должны с ее помощью получить 44 100 равномерно распределенных значений для передачи в PyGame. Процесс извлечения значений функции называется *выборкой*, поэтому напишем функцию `sample(f,start,end,count)`, извлекающую указанное количество значений заданной функции $f(t)$ в диапазоне значений аргумента t от `start` до `end`. Затем сможем вызвать `sample(make_sinusoid,0,1,44100)`, чтобы получить массив из 44 100 значений для передачи в PyGame и услышать, как звучит синусоидальная волна.

13.3.2. Изменение частоты синусоиды

В качестве первого примера создадим синусоиду с частотой 2, то есть функцию, имеющую форму синусоидального графика, но дважды повторяющуюся между нулем и единицей. Период синусоидальной функции равен 2π, поэтому для двукратного повторения потребуется 4π единиц (рис. 13.12).

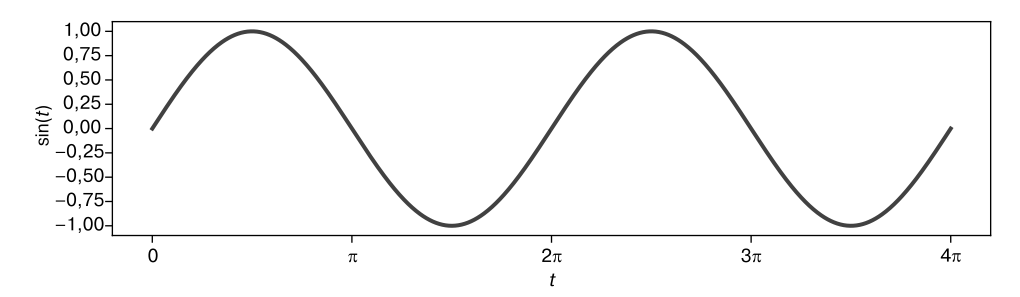


Рис. 13.12. Синусоидальная функция, дважды повторяющаяся между $t = 0$ и $t = 4\pi$

Чтобы получить два периода графика синусоидальной функции, следует обеспечить передачу ей значений от 0 до 4π, но нужно, чтобы входная переменная t менялась в диапазоне от 0 до 1. Для этого мы можем использовать функцию $\sin(4\pi t)$. Для диапазона от $t = 0$ до $t = 1$ функция синуса получит все значения от 0 до 4π. График $\sin(4\pi t)$, изображенный на рис. 13.13, выглядит точно так же, как график на рис. 13.12, но имеет два полных периода, сжатых в интервал от 0,0 до 1,0.

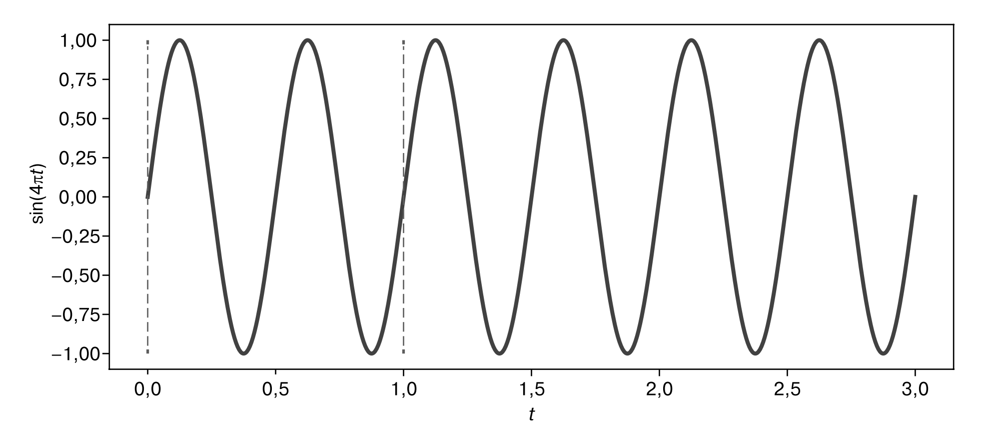


Рис. 13.13. График функции $\sin(4\pi t)$ имеет синусоидальную форму, дважды повторяющуюся на каждом единичном интервале значений t, что соответствует частоте 2 Гц

Период функции $\sin(4\pi t)$ равен 1/2, а не 2π, соответственно, коэффициент сжатия равен 4π. То есть исходный период имел протяженность 2π, а сжатый — в 4π раз короче. В общем случае для любой константы k функция вида $f(t) = \sin(kt)$ имеет период, сжатый в k раз, — до $2\pi/k$. Соответственно, частота увеличивается в k раз с начального значения $1/(2\pi)$ до $k/2\pi$.

Чтобы получить синусоидальную функцию с частотой 441, следует выбрать значение $k = 441 \cdot 2\pi$. Это даст нам частоту

$$\frac{441 \cdot 2\pi}{2\pi} = 441.$$

В отличие от частоты, амплитуду синусоиды увеличить намного проще — достаточно лишь умножить результат синусоидальной функции на константу, и амплитуда увеличится на эту константу. Теперь у нас есть все что нужно для определения функции `make_sinusoid`:

```
def make_sinusoid(frequency,amplitude):
    def f(t):
        return amplitude * sin(2*pi*frequency*t)
    return f
```

Определение f(t) — синусоидальной функции (к строке `def f(t):`)

Аргумент умножается на 2π и значение частоты, а затем результат функции синуса умножается на коэффициент амплитуды (к строке `return amplitude * sin(2*pi*frequency*t)`)

Мы можем проверить свой код, например, определив синусоидальную функцию с частотой 5 и амплитудой 4 и нарисовав ее график (рис. 13.14) в диапазоне от $t = 0$ до $t = 1$:

```
>>> plot_function(make_sinusoid(5,4),0,1)
```

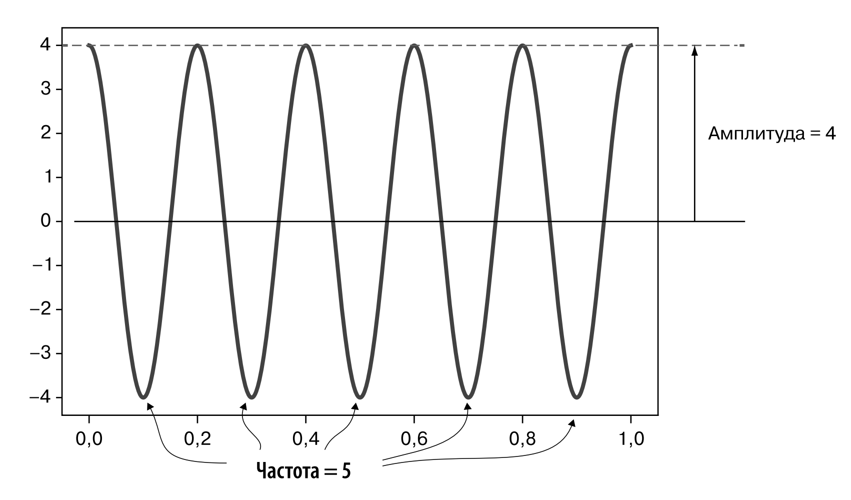


Рис. 13.14. График make_sinusoid(5,4) имеет высоту (амплитуду) 4 и 5 повторений периода в диапазоне от $t = 0$ до $t = 5$, соответственно, частота полученной функции равна 5

Далее мы создадим звуковую волну с частотой 441 Гц и амплитудой 8000, полученную вызовом `make_sinusoid(441,8000)`.

13.3.3. Выборка и воспроизведение звуковой волны

Чтобы воспроизвести звуковую волну, упомянутую в предыдущем разделе, нужно выбрать ее значения и поместить их в массив для передачи библиотеке PyGame. Сделаем это:

```
sinusoid = make_sinusoid(441,8000)
```

Функция `sinusoid` в диапазоне от $t = 0$ до $t = 1$ представляет звуковую волну с продолжительностью звучания 1 с. Теперь мы должны выбрать 44 100 значений t, равномерно распределенных в интервале между 0 и 1, и последовательно передать их функции `sinusoid(t)`.

Для этого можно использовать функцию `np.arange` из библиотеки NumPy, которая возвращает числа, равномерно распределенные на заданном интервале. Например, `np.arange(0,1,0.1)` даст 10 значений в диапазоне от 0 до 1, последовательно увеличивающихся на 0,1 единицы интервала:

```
>>> np.arange(0,1,0.1)
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

В нашем случае нужно получить 44 100 значений времени от 0 до 1, последовательно увеличивающихся на 1/44 100 единицы:

```
>>> np.arange(0,1,1/44100)
array([0.00000000e+00, 2.26757370e-05, 4.53514739e-05, ...,
       9.99931973e-01, 9.99954649e-01, 9.99977324e-01])
```

Далее к каждому элементу этого массива нужно применить функцию `sinusoid`, чтобы в результате создать еще один массив NumPy. Функция `np.vectorize(f)` принимает функцию `f` и создает новую функцию, которая выполняет одну и ту же операцию с *каждым* элементом массива, то есть `np.vectorize(sinusoid)(arr)` применит функцию `sinusoid` к каждому элементу массива.

Мы почти закончили выборку значений функции. Осталось лишь преобразовать полученные результаты в 16-битные целые значения с использованием метода `astype` массивов NumPy. Объединив эти шаги, получаем следующую функцию выборки:

```
def sample(f,start,end,count):          ◄ Входные данные: f — функция для выборки, начало и конец диапазона и количество значений, которое требуется получить
    mapf = np.vectorize(f)              ◄ Создать версию f, которую можно применить к массиву NumPy
    ts = np.arange(start,end,(end-start)/count)   ◄ Создать массив значений для передачи функции, равномерно распределенных в заданном диапазоне
    values = mapf(ts)                   ◄ Применить функцию к каждому значению в массиве NumPy
    return values.astype(np.int16)      ◄ Преобразовать получившиеся значения в 16-битные целые числа и вернуть их
```

Вооруженные этой функцией, мы можем услышать, как звучит синусоидальная волна с частотой 441 Гц:

```
sinusoid = make_sinusoid(441,8000)
arr = sample(sinusoid, 0, 1, 44100)
sound = pygame.sndarray.make_sound(arr)
sound.play()
```

Если воспроизвести ее, а затем прямоугольную волну с частотой 441 Гц, то можно заметить, что обе воспроизводят одну и ту же ноту, другими словами, обе дают звук одинаковой высоты. Однако качество звука сильно различается: синусоидальная волна дает гораздо более мягкий звук. Он звучит почти как флейта, а не как дребезжащие звуки из старой видеоигры. Это качество звука называется *тембром*.

В оставшейся части главы мы сосредоточимся на звуковых волнах, построенных из комбинаций синусоид. Оказывается, при правильном сочетании можно получить волну любой формы волны и, следовательно, любой желаемый тембр.

13.3.4. Упражнения

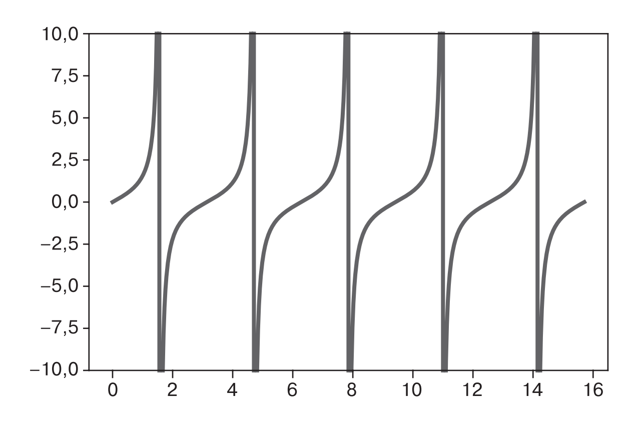
Упражнение 13.2. Постройте график тангенциальной функции $\tan(t) = \sin(t)/\cos(t)$. Какой период она имеет?

Решение. Тангенциальная функция уходит в бесконечность в каждом периоде, поэтому ее можно построить только в ограниченном диапазоне значений y:

```
from math import tan
plot_function(tan,0,5*pi)
plt.ylim(-10,10)
```

Ограничить размер окна графика по вертикали диапазоном $-10 < y < 10$

График функции tan(x), которая является периодической, выглядит таким образом:



Поскольку функция tan (t) зависит только от значений cos (t) и sin (t), она должна повторяться по крайней мере каждые 2π единицы. На самом деле она повторяется *дважды* через каждые 2π единицы — на графике ясно видно, что период этой функции равен π.

Упражнение 13.3. Определите частоту и период функции sin $(3\pi t)$? Какой период?

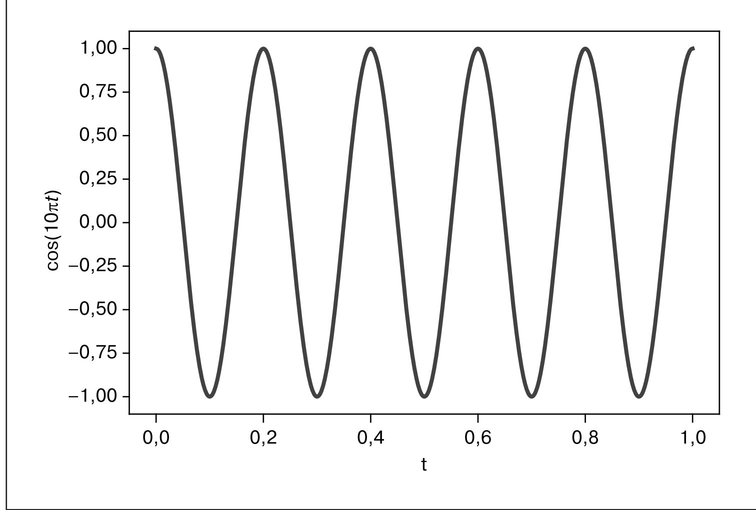
Решение. Частота sin (t) равна $1/(2\pi)$, а умножение аргумента t на 3π увеличивает эту частоту в 3π раза. Соответственно, частота составляет $(3\pi)/(2\pi) = 3/2$. Период является обратной величиной и равен 2/3.

Упражнение 13.4. Найдите значение k, при котором частота cos (kt) равна 5. Нарисуйте график получившейся функции cos (kt) в диапазоне от нуля до единицы и покажите, что она повторяется 5 раз.

Решение. Частота cos (t) по умолчанию равна $1/2\pi$, поэтому частота cos (kt) равна $k/2\pi$. Чтобы получить частоту, равную 5, нужно, чтобы $k = 10\pi$. Соответствующая функция будет определяться как cos $(10\pi t)$:

```
>>> plot_function(lambda t: cos(10*pi*t),0,1)
```

Вот график этой функции, на котором видно, что функция повторяется 5 раз в диапазоне значений от $t = 0$ до $t = 1$.



13.4. ОБЪЕДИНЕНИЕ ЗВУКОВЫХ ВОЛН

В главе 6 мы узнали, что с функциями можно обращаться как с векторами: их можно складывать или умножать на скаляры и получать новые функции. Создавая линейные комбинации функций, определяющих звуковые волны, можно генерировать новые интересные звуки.

Самый простой способ объединить две звуковые волны — выбрать последовательности значений обеих, а затем сложить соответствующие элементы двух массивов. Начнем с того, что напишем код, складывающий выборки звуковых волн разных частот и получающий результат, который звучит подобно музыкальному аккорду, как если бы вы одновременно играли на нескольких струнах гитары.

После этого мы сможем реализовать более интересный пример, в котором сложим вместе несколько десятков синусоидальных звуковых волн разных частот в заданную линейную комбинацию, получив результат, звучащий как прямоугольная волна, которую мы видели раньше.

13.4.1. Сложение выборок звуковых волн для получения аккорда

Массивы NumPy можно складывать с помощью обычного оператора +, что упрощает нашу задачу. Вот небольшой пример, показывающий, что при сложении массивов NumPy производится сложение соответствующих элементов массивов, в результате чего получается новый массив:

```
>>> np.array([1,2,3]) + np.array([4,5,6])
array([5, 7, 9])
```

Выясняется, что сумма двух звуковых волн дает такой же звук, как если бы вы воспроизводили обе волны сразу. Вот две выборки — из синусоид с частотой 441 Гц и 551 Гц:

```
sample1 = sample(make_sinusoid(441,8000),0,1,44100)
sample2 = sample(make_sinusoid(551,8000),0,1,44100)
```

Если запустить воспроизведение первой выборки и тут же — второй, то PyGame воспроизведет два звука почти одновременно. Запустив следующий код, вы должны услышать аккорд, состоящий из двух разных музыкальных нот. Если запустить любую из двух последних строк, вы услышите одну из двух отдельных нот:

```
sound1 = pygame.sndarray.make_sound(sample1)
sound2 = pygame.sndarray.make_sound(sample2)
sound1.play()
sound2.play()
```

Теперь, используя NumPy, мы можем сложить две выборки, чтобы создать и воспроизвести новый звук с помощью PyGame. При сложении `sample1` и `sample2`

создается новый массив длиной 44 100, содержащий суммы элементов из `sample1` и `sample2`. Если воспроизвести результат, он будет звучать точно так же, как предыдущий пример:

```
chord = pygame.sndarray.make_sound(sample1 + sample2)
chord.play()
```

13.4.2. Изображение графика суммы двух звуковых волн

Посмотрим, как выглядит график суммы звуковых волн. Возьмем для примера первые 400 точек из `sample1` (441 Гц) и `sample2` (551 Гц). На рис. 13.15 можно видеть, что выбранный период времени охватывает четыре периода звуковой волны, соответствующей первой выборке, и пять периодов звуковой волны, соответствующей второй выборке.

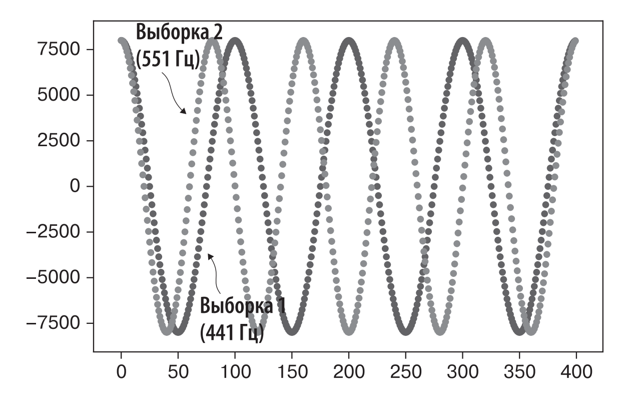


Рис. 13.15. График с первыми 400 точками из выборок sample1 и sample2

Может показаться неожиданным, что сложение `sample1` и `sample2` не дает синусоиды, даже при том что складываются две синусоиды. Вместо этого последовательность `sample1 + sample2` описывает волну с меняющейся амплитудой. На рис. 13.16 показано, как выглядит сумма.

Внимательно рассмотрим процедуру сложения, чтобы понять, как получилась такая форма волны. В районе 85-й точки обе волны имеют большие положительные значения, поэтому 85-я точка суммы тоже имеет большое положительное значение. Около 350-й точки обе волны имеют большие отрицательные значения, как и их сумма. Когда две волны совпадают, их сумма получается еще больше, а звук — громче. Этот эффект называется *конструктивной интерференцией*.

На рис. 13.17 можно видеть интересный эффект в районе 200-й точки, где значения волн противоположны. Например, соответствующее значение в `sample1` большое положительное, а значение в `sample2` — большое отрицательное.

Это приводит к тому, что их сумма оказывается близка к нулю, хотя значения в самих выборках далеки от нуля. Эффект, когда две волны гасят друг друга подобным образом, называется *деструктивной интерференцией*.

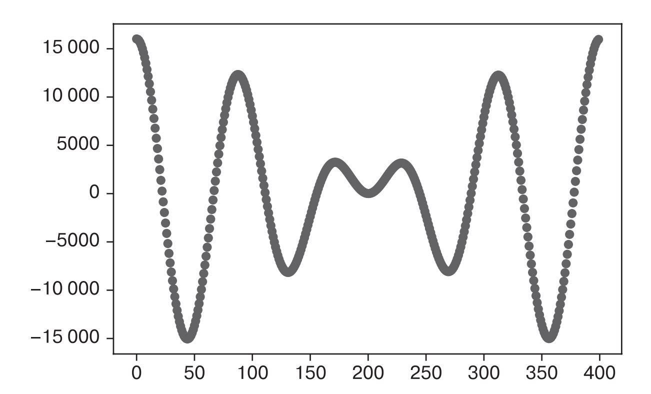


Рис. 13.16. График суммы двух волн, sample1 + sample2

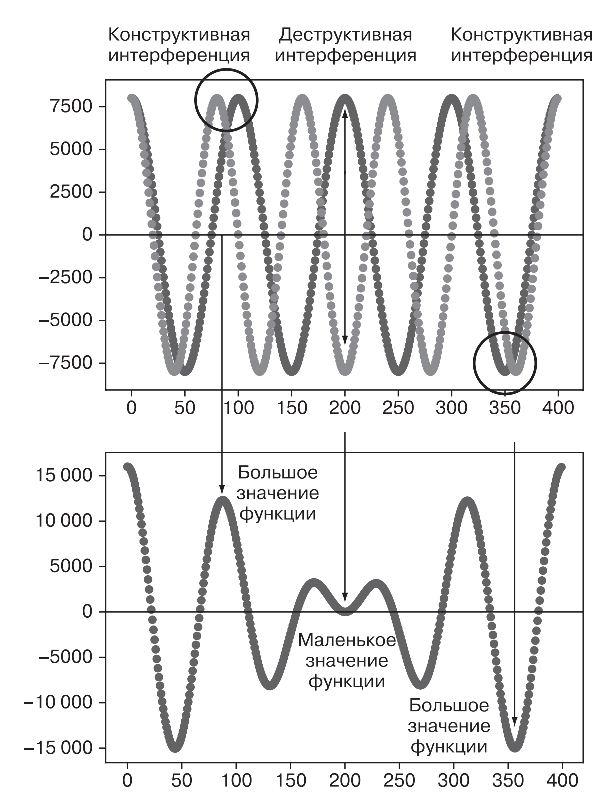


Рис. 13.17. Абсолютная величина суммарной волны велика при конструктивной интерференции и мала при деструктивной интерференции

Поскольку волны имеют разные частоты, они то синхронизируются, то рассинхронизируются друг с другом, чередуя созидательную и деструктивную интерференцию. Как следствие, сумма волн не является синусоидой и меняет амплитуду с течением времени. На рис. 13.17 два графика показаны друг под другом, чтобы сделать более наглядной взаимосвязь между двумя выборками и их суммой.

Как видите, относительные частоты складываемых синусоид влияют на форму получаемого графика. Далее я покажу вам еще более экстремальный пример построения линейной комбинации из нескольких десятков синусоидальных функций.

13.4.3. Построение линейной комбинации синусоид

Начнем с создания большого набора синусоид разных частот. Мы можем составить список (сколь угодно длинный) функций синуса, начиная с $\sin(2\pi t)$, $\sin(4\pi t)$, $\sin(6\pi t)$, $\sin(8\pi t)$ и т. д. Эти функции имеют частоты 1, 2, 3, 4 и т. д. И точно так же можем составить список функций косинуса: $\cos(2\pi t)$, $\cos(4\pi t)$, $\cos(6\pi t)$, $\cos(8\pi t)$... с частотами 1, 2, 3, 4 и т. д. Суть здесь в том, что, имея такое количество различных частот, мы можем создавать звуковые волны самых разных форм, взяв линейные комбинации этих функций. По причинам, которые мы увидим позже, я также включу в линейную комбинацию постоянную функцию $f(x) = 1$. Для случая с самой высокой частотой N линейную комбинацию синусов, косинусов и постоянной функции можно представить, как показано на рис. 13.18.



Рис. 13.18. Линейная комбинация функций синуса и косинуса

Эта линейная комбинация представляет ряд Фурье, и сама является функцией переменной t. Она задается $2N + 1$ числами: постоянным членом a_0, коэффициентами от a_1 до a_N для косинуса и коэффициентами от b_1 до b_N для синуса. Чтобы вычислить функцию, можно подставить заданное значение t в каждый синус и косинус и сложить результаты. Реализуем это на Python, чтобы потом протестировать несколько различных рядов Фурье.

Функция `fourier_series` принимает единственную константу a_0 и списки a и b, содержащие коэффициенты a_1, ..., a_N и b_1, ..., b_N соответственно. Она будет правильно работать, даже если получит списки разной длины, интерпретируя отсутствующие коэффициенты как равные нулю. Обратите внимание на то, что частоты синуса и косинуса начинаются с единицы, тогда как индексация массивов в Python — с нуля, поэтому $(n + 1)$ — это частота, соответствующая коэффициенту с индексом n в любом из массивов:

```
def const(n):
    return 1

def fourier_series(a0,a,b):
    def result(t):
        cos_terms = [an*cos(2*pi*(n+1)*t)
                     for (n,an) in enumerate(a)]
        sin_terms = [bn*sin(2*pi*(n+1)*t)
                     for (n,bn) in enumerate(b)]
        return a0*const(t) + \
            sum(cos_terms) + sum(sin_terms)
    return result
```

Постоянная функция, всегда возвращающая 1

Вычислить все косинусы, умножить на соответствующие коэффициенты

Вычислить все синусы, умножить на соответствующие коэффициенты

Сложить оба результата с постоянным коэффициентом a0, умноженным на значение постоянной функции (1)

Вот пример вызова этой функции с $b_4 = 1$, $b_5 = 1$ и всеми остальными константами, равными 0. Это очень короткий ряд Фурье, $\sin(8\pi t) + \sin(10\pi t)$, график которого показан на рис. 13.19. Поскольку соотношение частот равно 4 : 5, результат должен получиться похожим на предыдущий график (см. рис. 13.17):

```
>>> f = fourier_series(0,[0,0,0,0,0],[0,0,0,1,1])
>>> plot_function(f,0,1)
```

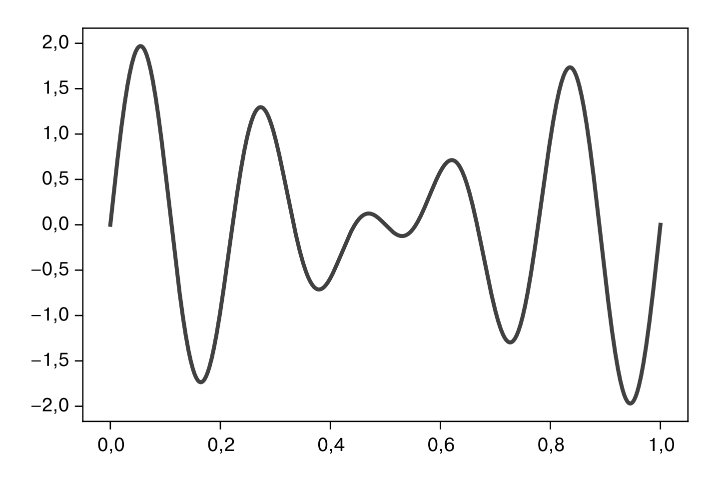


Рис. 13.19. График ряда Фурье $\sin(8\pi t) + \sin(10\pi t)$

Это не только хорошая проверка для нашей функции, но еще и отличная демонстрация возможностей рядов Фурье. Далее попробуем построить ряд Фурье с бóльшим количеством членов.

13.4.4. Построение знакомых функций с помощью синусоид

Создадим ряд Фурье, в котором по-прежнему нет постоянных членов и косинусов, но гораздо больше синусов. В частности, используем следующую последовательность значений для b_1, b_2, b_3 и т. д.:

$$b_1 = \frac{4}{\pi};\ b_2 = 0;\ b_3 = \frac{4}{3\pi};\ b_4 = 0;\ b_5 = \frac{4}{5\pi};\ b_6 = 0;\ b_7 = \frac{4}{7\pi}\ldots,$$

где $b_n = 0$ для четных n и $b_n = 4/(n\pi)$, для нечетных n. Это дает нам основу для построения ряда Фурье с любым количеством членов. Например, первый ненулевой член равен

$$\frac{4}{\pi}\sin(2\pi t),$$

и с добавлением следующего члена ряд превращается в

$$\frac{4}{\pi}\sin(2\pi t) + \frac{4}{3\pi}\sin(6\pi t).$$

Далее приводится код, реализующий этот ряд Фурье:

```
>>> f1 = fourier_series(0,[],[4/pi])
>>> f3 = fourier_series(0,[],[4/pi,0,4/(3*pi)])
>>> plot_function(f1,0,1)
>>> plot_function(f3,0,1)
```

а на рис. 13.20 показаны графики этих двух функций.

Используя генератор списков, можно составить гораздо более длинный список коэффициентов b_n и программно построить ряд Фурье. Мы можем оставить список коэффициентов при косинусах пустым, и он будет интерпретироваться так же, как если бы все коэффициенты были заданы равными 0:

```
b = [4/(n * pi)
     if n%2 != 0 else 0 for n in range(1,10)]
f = fourier_series(0,[],b)
```

Перечисляет значения bn = 4/nπ для нечетных значений n и bn = 0 — для четных

Этот список охватывает диапазон $1 \le n < 10$, поэтому ненулевые значения получают коэффициенты b_1, b_3, b_5, b_7 и b_9. При таких условиях график ряда выглядит, как показано на рис. 13.21.

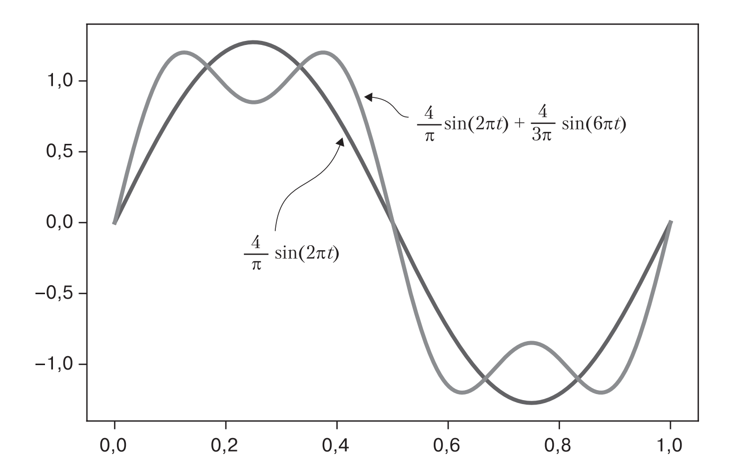


Рис. 13.20. Графики функций, образованных одним и двумя первыми членами ряда Фурье

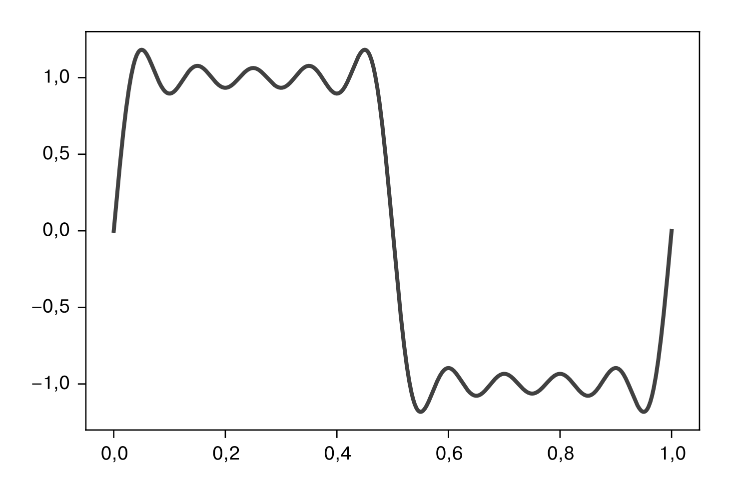


Рис. 13.21. Сумма первых пяти ненулевых членов ряда Фурье

Это интересный образец конструктивной и деструктивной интерференции! Около $t = 0$ и $t = 1$ все синусоидальные функции одновременно увеличиваются, а около $t = 0,5$ — уменьшаются. Эта конструктивная интерференция является доминирующим эффектом, а чередование конструктивной и деструктивной интерференции делает график относительно плоским в других областях. При n до 19 ряд Фурье имеет 10 ненулевых членов, и их взаимовлияние выглядит еще поразительнее (рис. 13.22):

```
>>> b = [4/(n * pi) if n%2 != 0 else 0 for n in range(1,20)]
>>> f = fourier_series(0,[],b)
```

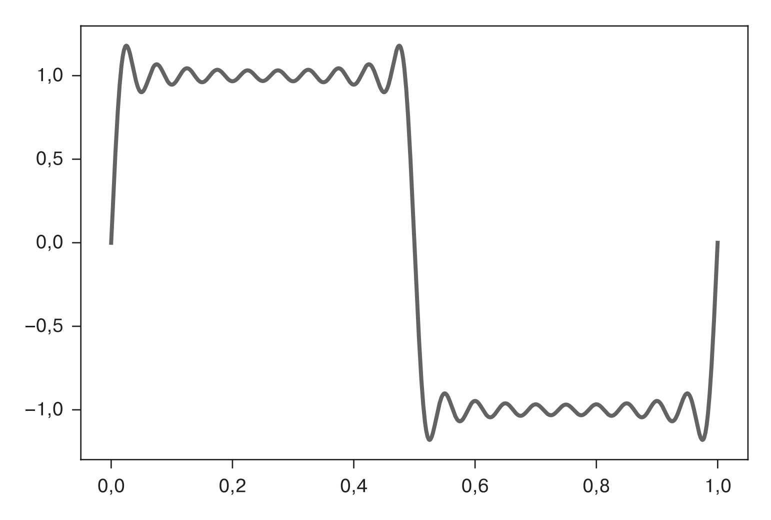


Рис. 13.22. Сумма первых 10 ненулевых членов ряда Фурье

Если увеличить n до 99, то мы получим сумму 50 синусоид, и функция станет почти плоской, за исключением нескольких больших скачков (рис. 13.23):

```
>>> b = [4/(n * pi) if n%2 != 0 else 0 for n in range(1,100)]
>>> f = fourier_series(0,[],b)
```

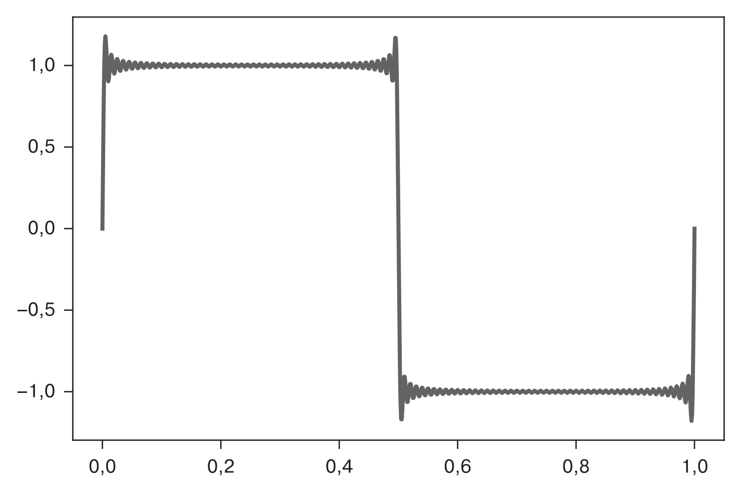


Рис. 13.23. График ряда Фурье при n = 99 дает практически плоский график, за исключением больших скачков в точках 0, 0,5 и 1

Если уменьшить масштаб, то можно увидеть, что этот ряд Фурье близок к прямоугольной волне, которую мы построили в начале главы (рис. 13.24).

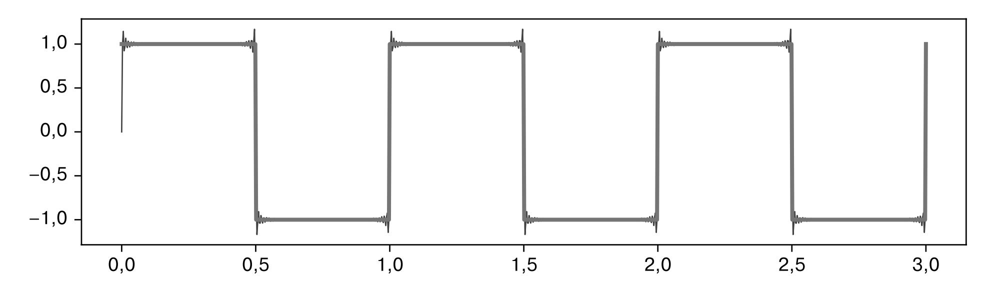


Рис. 13.24. Первые 50 ненулевых членов ряда Фурье создают график, близкий к прямоугольной волне, подобно первой функции, с которой мы познакомились в этой главе

В последнем примере мы фактически построили аппроксимацию прямоугольной волновой функции, использовав линейную комбинацию синусоид. Это кажется невероятным, потому что все синусоиды в ряду Фурье округлые и гладкие, а прямоугольная волна плоская, с резкими переходами. В заключение главы мы посмотрим, как реконструировать эту аппроксимацию, взяв любую периодическую функцию и восстановив коэффициенты ряда Фурье, аппроксимирующего ее.

13.4.5. Упражнения

Упражнение 13.5. Мини-проект. Создайте версию ряда Фурье, воссоздающего прямоугольную волну, чтобы ее частота составляла 441 Гц, затем получите выборку и убедитесь, что она не только выглядит, но и звучит как прямоугольная волна.

13.5. РАЗЛОЖЕНИЕ ЗВУКОВОЙ ВОЛНЫ В РЯД ФУРЬЕ

Наша последняя цель — взять произвольную периодическую функцию, например прямоугольную волну, и придумать, как представить ее (по крайней мере приближенно) в виде линейной комбинации синусоидальных функций. Проще говоря, мы должны научиться разбивать любую звуковую волну на комбинацию чистых нот. В качестве базового примера рассмотрим звуковую волну, представляющую аккорд, и определим, из каких нот он состоит. Вообще говоря, на музыкальные ноты можно разбить любой звук: человеческую речь, лай собаки или звук работающего двигателя автомобиля. Это утверждение основывается на несколько элегантных математических идеях, и теперь у нас есть все необходимое для их понимания.

Процесс разложения функции в ряд Фурье аналогичен записи вектора в виде линейной комбинации базисных векторов, как мы делали в части I. Взяв эту аналогию за основу, будем работать в векторном пространстве функций и интерпретировать функции, подобные прямоугольной волне, как целевые. Затем представим базис в виде набора функций sin $(2\pi t)$, sin $(4\pi t)$, sin $(6\pi t)$ и т. д. В разделе 13.3 мы аппроксимировали прямоугольную волну линейной комбинацией, начинающейся с

$$\frac{4}{\pi}\sin(2\pi t)+\frac{4}{3\pi}\sin(6\pi t)+\ldots$$

Два базисных вектора, sin $(2\pi t)$ и sin $(6\pi t)$, можно изобразить как два перпендикулярных направления в бесконечномерном пространстве функций со множеством других направлений, определяемых другими базисными векторами. Прямоугольная волна имеет составляющую длины $4/\pi$ в направлении sin $(2\pi t)$ и составляющую длины $4/3\pi$ в направлении sin $(6\pi t)$. Это первые две координаты прямоугольной волны из бесконечного списка координат в этом базисе (рис. 13.25).

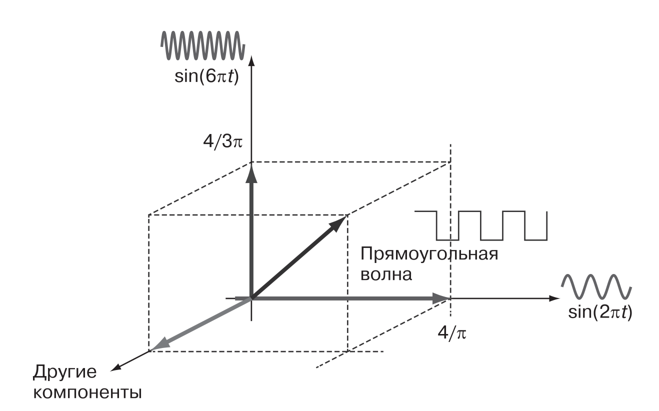


Рис. 13.25. Прямоугольную волну можно рассматривать как вектор в пространстве функций с длиной компоненты $4/\pi$ в направлении sin $(2\pi t)$ и $4/3\pi$ в направлении sin $(6\pi t)$. Прямоугольная волна включает бесконечное число других компонент

Мы можем написать функцию `fourier_coefficients(f,N)`, которая принимает периодическую функцию f с периодом, равным единице, и число N желаемых коэффициентов. Она будет интерпретировать постоянную функцию, а также функции cos $(2n\pi t)$ и sin $(2n\pi t)$ в диапазоне $1 \le n < N$ как направления в векторном пространстве функций и находить компоненты f в этих направлениях. Результатом функции будут коэффициент a_0, представляющий постоянную функцию, и списки коэффициентов Фурье $a_1, a_2, ..., a_N$ и $b_1, b_2, ..., b_N$.

13.5.1. Поиск компонент вектора с помощью внутреннего произведения

В главе 6 мы видели, как складывать векторы и умножать их на скаляры с помощью функций по аналогии с операциями с двух- и трехмерными векторами. Еще один инструмент, который нам понадобится, — это аналогия скалярного произведения. Скалярное произведение — один из примеров *внутреннего произведения*, способа перемножения двух векторов для получения скаляра, оценивающего их сонаправленность.

На мгновение вернемся в трехмерный мир и посмотрим, как использовать скалярное произведение для поиска компонент трехмерного вектора, а затем применим тот же прием, чтобы найти компоненты функции в базисе синусоидальных функций. Предположим, что наша цель — найти компоненты вектора $\mathbf{v} = (3, 4, 5)$ в терминах стандартных базисных векторов, $\mathbf{e}_1 = (1, 0, 0)$, $\mathbf{e}_2 = (0, 1, 0)$ и $\mathbf{e}_3 = (0, 0, 1)$. Решение настолько очевидно, что мы никогда не задумывались над ним. Компоненты 3, 4 и 5 соответственно — вот что означают координаты (3, 4, 5)!

Сейчас я покажу другой способ определения компонент $\mathbf{v} = (3, 4, 5)$ с помощью скалярного произведения. Для трехмерных векторов это излишне, потому что у нас уже есть ответ, но этот прием пригодится для векторов-функций. Обратите внимание на то, что каждое скалярное произведение $\mathbf{v}$ со стандартным базисным вектором возвращает одну из компонент:

$$\mathbf{v} \cdot \mathbf{e}_1 = (3, 4, 5) \cdot (1, 0,0) = 3 + 0 + 0 = 3;$$

$$\mathbf{v} \cdot \mathbf{e}_2 = (3, 4, 5) \cdot (0, 1,0) = 0 + 4 + 0 = 4;$$

$$\mathbf{v} \cdot \mathbf{e}_3 = (3, 4, 5) \cdot (0, 0,1) = 0 + 0 + 5 = 5.$$

Эти скалярные произведения прямо говорят, как записать v в виде линейной комбинации стандартного базиса: $\mathbf{v} = 3\mathbf{e}_1 + 4\mathbf{e}_2 + 5\mathbf{e}_3$. Но будьте внимательны — это определение верно только потому, что скалярное произведение согласуется с определениями длин и углов. Любая пара перпендикулярных стандартных базисных векторов имеет нулевое скалярное произведение:

$$\mathbf{e}_1 \cdot \mathbf{e}_2 = \mathbf{e}_2 \cdot \mathbf{e}_3 = \mathbf{e}_3 \cdot \mathbf{e}_1 = 0.$$

А скалярные произведения стандартных базисных векторов на самих себя дают их длины (в квадрате), равные 1:

$$\mathbf{e}_1 \cdot \mathbf{e}_1 = \mathbf{e}_2 \cdot \mathbf{e}_2 = \mathbf{e}_3 \cdot \mathbf{e}_3 = |\mathbf{e}_1|^2 = |\mathbf{e}_2|^2 = |\mathbf{e}_3|^2 = 1.$$

Кроме того, согласно скалярному произведению, ни один из стандартных базисных векторов не имеет компонент в направлениях других стандартных базисных векторов и каждый стандартный базисный вектор имеет компоненту 1 в своем собственном направлении. Для использования внутреннего произведения при вычислении компонент функций необходимо, чтобы наш базис обладал такими же свойствами. Иначе говоря, базисные функции, такие как sin ($2\pi t$), cos ($2\pi t$) и т. д., должны быть перпендикулярны друг другу и иметь длину 1. Далее мы определим внутреннее произведение для функций и проверим эти факты.

13.5.2. Определение внутреннего произведения периодических функций

Предположим, что $f(t)$ и $g(t)$ — две функции, определенные на интервале от $t = 0$ до $t = 1$ и повторяющиеся через каждую единицу t. Скалярное произведение f и g можно записать как $\langle f, g\rangle$ и определить его через определенный интеграл:

$$\langle f, g\rangle = 2\cdot\int_0^1 f(t)g(t)dt.$$

Реализуем эту формулу на Python, аппроксимировав интеграл суммой Римана, как делали в главе 8, чтобы вы могли увидеть, что это внутреннее произведение работает подобно знакомому скалярному произведению. Следующая сумма Римана по умолчанию равна 1000 временных шагов:

```
def inner_product(f,g,N=1000):
    dt = 1/N
    return 2*sum([f(t)*g(t)*dt
                  for t in np.arange(0,1,dt)])
```

Размер dt по умолчанию составляет 1/1000 = 0,001

Вклад каждого временного шага в интеграл равен f(t) * g(t) * dt. Результат интеграла умножается на 2 согласно формуле

Как и скалярное произведение, эта интегральная аппроксимация вычисляется как сумма произведений значений из входных векторов. Это не сумма произведений координат, а сумма произведений значений функций. Значения функций можно рассматривать как выборку из бесконечного множества координат, а это внутреннее произведение — как своего рода бесконечное скалярное произведение по этим координатам.

Рассмотрим, как вычисляется внутреннее произведение. Для удобства определим вспомогательные функции, вычисляющие n-е функции синуса и косинуса в нашем базисе, а затем протестируем их с помощью функции `inner_product`. Эти функции похожи на упрощенные версии функции `make_sinusoid` из раздела 13.3.2:

```
def s(n):
    def f(t):
        return sin(2*pi*n*t)
    return f
```

s(n) принимает целое число n и возвращает функцию sin(2nπt)

```
def c(n):
    def f(t):
        return cos(2*pi*n*t)
    return f
```

s(n) принимает целое число n и возвращает функцию cos(2nπt)

Скалярное произведение двух трехмерных векторов, таких как (1, 0, 0) и (0, 1, 0), дает ноль, подтверждая, что они перпендикулярны. Внутреннее произведение показывает, что все пары базисных функций взаимно перпендикулярны (почти), например:

```
>>> inner_product(s(1),c(1))
4.2197487366314734e-17
>>> inner_product(s(1),s(2))
-1.4176155163484784e-18
>>> inner_product(c(3),s(10))
-1.7092447249233977e-16
```

Эти числа чрезвычайно близки к нулю, подтверждая, что sin $(2\pi t)$ и cos $(2\pi t)$ взаимно перпендикулярны, а sin $(2\pi t)$ и sin $(4\pi t)$ строго перпендикулярны, так же как cos $(6\pi t)$ и cos $(20\pi t)$. Используя точные формулы интегрирования, которые мы здесь не будем рассматривать, можно *доказать*, что для любых целых чисел n и m

$$\left\langle \sin(2n\pi t), \cos(2m\pi t) \right\rangle = 0,$$

для любых различных целых чисел n и m

$$\left\langle \sin(2n\pi t), \sin(2m\pi t) \right\rangle = 0$$

и

$$\left\langle \cos(2n\pi t), \cos(2m\pi t) \right\rangle = 0.$$

То есть согласно результату этого внутреннего произведения все наши синусоидальные базисные функции взаимно перпендикулярны, ни у одной нет компоненты в направлении другой. Еще нужно проверить, подразумевает ли внутреннее произведение, что наши базисные векторы имеют компоненты с длиной 1 в своих собственных направлениях. Действительно, это утверждение выглядит верным в пределах числовой ошибки:

```
>>> inner_product(s(1),s(1))
1.0000000000000002
>>> inner_product(c(1),c(1))
0.9999999999999999
>>> inner_product(c(3),c(3))
1.0
```

Мы не будем останавливаться на доказательствах, но хочу вас заверить, что с помощью интегральных формул можно прямо доказать, что для любого целого числа n выполняются равенства

$$\left\langle \sin(2n\pi t), \sin(2n\pi t) \right\rangle = 1$$

и

$$\left\langle \cos(2n\pi t), \cos(2n\pi t) \right\rangle = 1.$$

Последнее, что нам нужно сделать, — включить в обсуждение постоянную функцию. Ранее я обещал, что объясню, зачем нужен постоянный член в ряду Фурье, и теперь могу дать первоначальное пояснение. Постоянная функция необходима для построения полного базиса функций, не включать ее было бы сродни исключению $\mathbf{e}_2$ из базиса трехмерного пространства и использованию только $\mathbf{e}_1$ и $\mathbf{e}_3$. В этом случае вы просто не сможете сконструировать некоторые функции на основе базисных векторов.

Любая постоянная функция перпендикулярна любой функции синуса и косинуса в нашем базисе, но мы должны выбрать такое значение постоянной функции, чтобы она имела компоненту 1 в своем собственном направлении. То есть если предположить, что постоянная функция реализована с именем `const(t)`, то вызов `inner_product(const,const)` должен возвращать 1. Правильное постоянное значение, которое должна возвращать функция `const`, равно $1/\sqrt{2}$ (в следующем упражнении вы сможете убедиться, что оно имеет смысл!):

```
from math import sqrt

def const(n):
    return 1 /sqrt(2)
```

Определив постоянную функцию, можно подтвердить, что она имеет правильные свойства:

```
>>> inner_product(const,s(1))
-2.2580204307905138e-17
>>> inner_product(const,c(1))
-3.404394821604484e-17
>>> inner_product(const,const)
1.0000000000000007
```

Теперь у нас есть все необходимое для поиска коэффициентов Фурье периодической функции. Эти коэффициенты — не что иное, как компоненты функции в определенном нами базисе.

13.5.3. Определение функции для поиска коэффициентов Фурье

В трехмерном примере мы видели, что скалярное произведение вектора **v** на базисный вектор $\mathbf{e}_i$ дает компоненту **v** в направлении $\mathbf{e}_i$. Тот же процесс используем для периодической функции f.

Коэффициенты a_n для $n \geq 1$ сообщают нам компоненты f в направлении базисной функции cos ($2n\pi t$). Они вычисляются как внутренние произведения f с этими базисными функциями:

$$a_n = \left\langle f, \cos\left(2n\pi t\right)\right\rangle, \textbf{ где } n \geq 1.$$

Точно так же каждый коэффициент b_n сообщает нам компоненту f в направлении базисной функции sin ($2n\pi t$), и его также можно вычислить с помощью скалярного произведения:

$$b_n = \left\langle f, \sin\left(2n\pi t\right)\right\rangle.$$

Наконец, число a_0 есть скалярное произведение f на постоянную функцию, значение которой равно $1/\sqrt{2}$. Все эти коэффициенты Фурье можно вычислить с помощью функций, написанных нами на Python, то есть мы готовы собрать функцию `fourier_coefficients`, которую собирались написать. Помните, что первый аргумент — это функция, которую мы хотим проанализировать, а второй аргумент — это максимальное количество синусоидальных и косинусоидальных составляющих:

```
def fourier_coefficients(f,N):
    a0 = inner_product(f,const)
    an = [inner_product(f,c(n))
          for n in range(1,N+1)]
    bn = [inner_product(f,s(n))
          for n in range(1,N+1)]
    return a0, an, bn
```

Постоянный член a_0 — внутреннее произведение f на постоянную базисную функцию

Коэффициенты a_n вычисляются как внутренние произведения f на cos(2nπt) для 1 < n < N + 1

Коэффициенты b_n вычисляются как внутренние произведения f на sin(2nπt) для 1 < n < N + 1

Для проверки передадим функции `fourier_coefficients` известный ряд Фурье и убедимся, что она возвращает известные коэффициенты:

```
>>> f = fourier_series(0,[2,3,4],[5,6,7])
>>> fourier_coefficients(f,3)
(-3.812922200197022e-15,
 [1.9999999999999887, 2.999999999999999, 4.0],
 [5.000000000000002, 6.000000000000001, 7.0000000000000036])
```

ПРИМЕЧАНИЕ

Чтобы входные и выходные данные соответствовали ненулевым постоянным членам, нужно пересмотреть функцию const и использовать $f(t) = 1/\sqrt{2}$ вместо $f(t) = 1$. См. упражнение 13.8.

Теперь, получив возможность автоматически вычислять коэффициенты Фурье, можно завершить исследование — построить несколько приближений Фурье для периодических функций интересной формы.

13.5.4. Поиск коэффициентов Фурье для прямоугольной волны

В предыдущем разделе мы видели, что все коэффициенты Фурье для прямоугольной волны равны нулю, за исключением коэффициентов b_n для нечетных значений n. То есть ряд Фурье строится как линейная комбинация функции $\sin(2n\pi t)$ для нечетных значений n. Для нечетного n коэффициент $b_n = 4/n\pi$. Тогда я не стал объяснять, почему коэффициенты имеют именно такой вид, но теперь мы можем проверить свою работу.

Чтобы создать прямоугольную волну, которая повторяется каждую единицу времени t, можно использовать выражение `t%1` на Python, вычисляющее дробную часть t. Поскольку, например, `2.3 % 1` равно `0.3`, а `0.3 % 1` равно `0.3`, функция, записанная в терминах `t % 1`, автоматически становится периодической с периодом 1. Прямоугольная волна имеет значение +1, когда `t % 1 < 0.5`, и −1 в противном случае:

```
def square(t):
    return 1 if (t%1) < 0.5 else -1
```

Найдем первые десять коэффициентов Фурье для этой прямоугольной волны. Для этого выполните инструкцию

```
a0, a, b = fourier_coefficients(square,10)
```

и вы увидите, что a_0 и все другие коэффициенты a имеют очень маленькие значения, как и все коэффициенты b с четными индексами. Значения b_1, b_3, b_5 и т. д. представлены элементами `b[0]`, `b[2]`, `b[4]` и т. д., потому что нумерация элементов массивов в Python начинается с нуля. Все они близки к ожидаемым значениям:

```
>>> b[0], 4/pi
(1.273235355942202, 1.2732395447351628)
>>> b[2], 4/(3*pi)
(0.4244006151333577, 0.4244131815783876)
>>> b[4], 4/(5*pi)
(0.2546269646514865, 0.25464790894703254)
```

Мы уже видели, что ряд Фурье с этими коэффициентами — это довольно точная аппроксимация прямоугольной волны. Завершим этот раздел еще двумя примерами функций, которых мы раньше не видели, и построим для них ряды Фурье, чтобы показать, что аппроксимация работает.

13.5.5. Коэффициенты Фурье для других волнообразных функций

Далее мы рассмотрим другие периодические функции, которые можно смоделировать с помощью преобразования Фурье. На рис. 13.26 показан график новой периодической функции интересной формы, которая называется *пилообразной волной*.

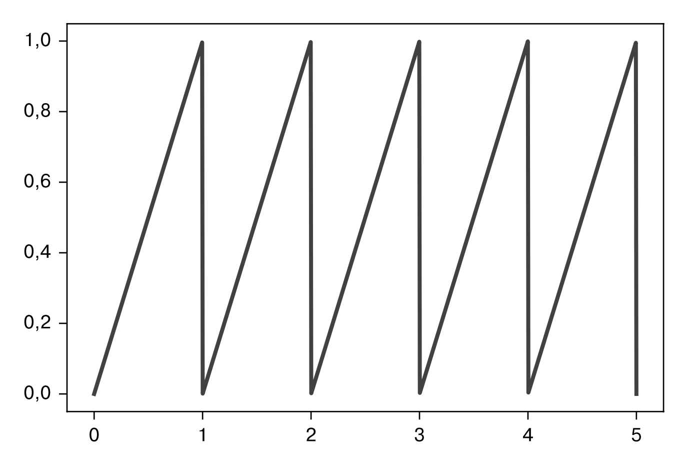


Рис. 13.26. Пять периодов пилообразной волны

На интервале от $t = 0$ до $t = 1$ пилообразная волна идентична функции $f(t) = t$, а затем повторяется через каждую единицу. На Python пилообразная функция определяется просто:

```
def sawtooth(t):
    return t%1
```

Чтобы увидеть аппроксимацию с использованием ряда Фурье с 10 синусоидальными и косинусоидальными членами, подставим коэффициенты Фурье непосредственно в функцию ряда Фурье и построим ее график рядом с пилообразным графиком, как показано на рис. 13.27. Как видите, аппроксимация получилась довольно точной:

```
>>> approx = fourier_series(*fourier_coefficients(sawtooth,10))
>>> plot_function(sawtooth,0,5)
>>> plot_function(approx,0,5)
```

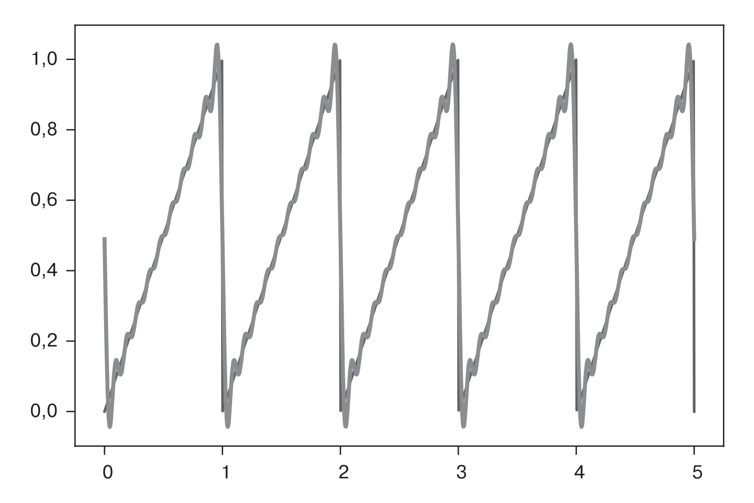


Рис. 13.27. Исходная пилообразная волна с графика на рис. 13.26 и ее аппроксимация рядом Фурье

И снова обратите внимание на то, насколько близко можно аппроксимировать функцию с острыми углами, используя только линейную комбинацию гладких синусоидальных и косинусоидальных функций. Эта функция имеет ненулевой постоянный коэффициент a_0. Он необходим, потому что эта функция имеет только значения выше нуля, а у функций синуса и косинуса есть отрицательные значения.

В качестве последнего примера рассмотрим еще одну функцию, определенную как `speedbumps(t)` в примерах исходного кода для этой книги. Ее график показан на рис. 13.28.

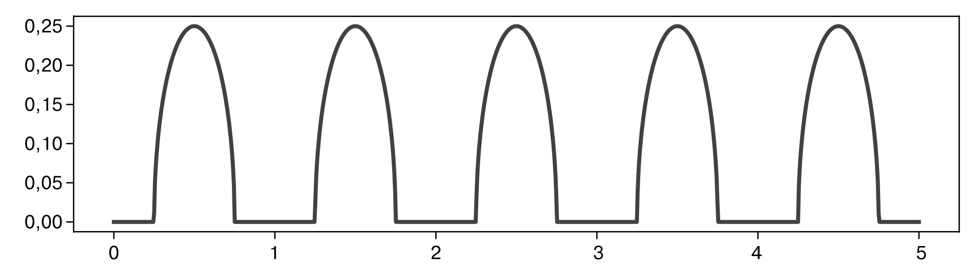


Рис. 13.28. Функция speedbumps(t), на графике которой плоские участки чередуются с округлыми

Реализация этой функции не особенно важна. Этот пример интересен тем, что получившийся ряд Фурье имеет ненулевые коэффициенты при функциях

косинуса и нулевые при функциях синуса. Даже с 10 членами мы получаем хорошее приближение. На рис. 13.29 показан график ряда Фурье с a_0 и 10 косинусоидальными членами (все коэффициенты b_n равны нулю).

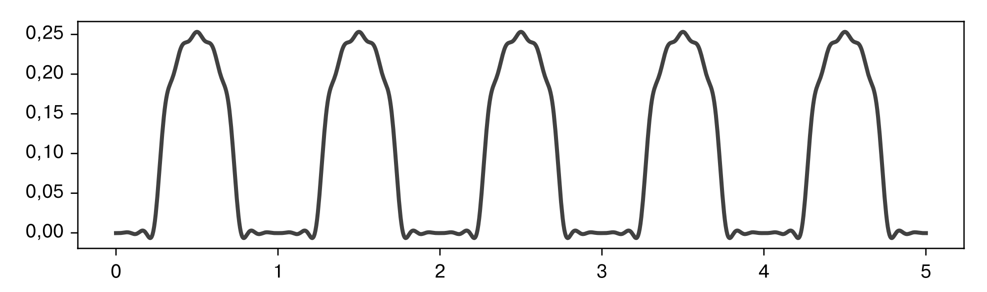


Рис. 13.29. Аппроксимация функции speedbumps(t) рядом Фурье с постоянным членом и 10 косинусными членами

На графике можно заметить некоторые колебания в этой аппроксимации, но при преобразовании в звук ряд Фурье довольно точно воспроизводит звучание исходной волны. Имея возможность преобразовывать сигналы всех форм в списки коэффициентов рядов Фурье, можно эффективно хранить и передавать аудиофайлы.

13.5.6. Упражнения

Упражнение 13.6. Векторы $\mathbf{u}_1 = (2, 0, 0)$, $\mathbf{u}_2 = (0, 1, 1)$ и $\mathbf{u}_3 = (1, 0, -1)$ образуют базис для $\mathbb{R}^3$. Для вектора $\mathbf{v} = (3, 4, 5)$ вычислите три скалярных произведения $a_1 = \mathbf{v} \cdot \mathbf{u}_1$, $a_2 = \mathbf{v} \cdot \mathbf{u}_2$ и $a_3 = \mathbf{v} \cdot \mathbf{u}_3$. Покажите, что $\mathbf{v}$ *не* равен $a_1\mathbf{u}_1 + a_2\mathbf{u}_2 + a_3\mathbf{u}_3$. Почему они не равны?

Решение. Скалярные произведения

$$a_1 = \mathbf{v} \cdot \mathbf{u}_1 = (3, 4, 5) \cdot (2, 0, 0) = 6;$$

$$a_2 = \mathbf{v} \cdot \mathbf{u}_2 = (3, 4, 5) \cdot (0, 1, 1) = 9;$$

$$a_3 = \mathbf{v} \cdot \mathbf{u}_3 = (3, 4, 5) \cdot (1, 0, -1) = -2.$$

Получается линейная комбинация $6 \cdot (2, 0, 0) + 9 \cdot (0, 1, 1) - 2 \cdot (1, 0, -1) = (16, 9, 2)$, которая не равна $(3, 4, 5)$. Этот подход не дает правильного результата, потому что длины этих базисных векторов не равны 1, а сами они не перпендикулярны друг другу.

Упражнение 13.7. Мини-проект. Предположим, что $f(t)$ — постоянная функция, то есть $f(t) = k$. Используйте интегральную формулу для вычисления скалярного произведения, чтобы найти значение k, при котором $\langle f, f \rangle = 1$. (Да, я уже говорил, что $k = 1/\sqrt{2}$, но проверьте, сможете ли вы сами получить это значение!)

Решение. Если $f(t) = k$, то $\langle f, f \rangle$ определяется интегралом:

$$2 \cdot \int_0^1 f(t) \cdot f(t) dt = 2 \cdot \int_0^1 k \cdot k\, dt = 2k^2.$$

(Площадь под постоянной функцией k^2 на интервале от 0 до 1 равна k^2.) Чтобы выражение $2k^2$ давало в результате 1, нужно, чтобы $k^2 = 1/1$ и, соответственно, $k = \sqrt{1/2} = 1/\sqrt{2}$.

Упражнение 13.8. Обновите функцию `fourier_series`, чтобы в качестве постоянной функции она использовала $f(t) = 1/\sqrt{2}$ вместо $f(t) = 1$.

Решение

Умножить коэффициент a0 на постоянную функцию $f(t) = 1 / \sqrt{2}$ в линейной комбинации и тем самым внести вклад $a_0 / \sqrt{2}$ в результат ряда Фурье независимо от значения t

```
def fourier_series(a0,a,b):
    def result(t):
        cos_terms = [an*cos(2*pi*(n+1)*t) for (n,an) in enumerate(a)]
        sin_terms = [bn*sin(2*pi*(n+1)*t) for (n,bn) in enumerate(b)]
        return a0/sqrt(2) + sum(cos_terms) + sum(sin_terms)
    return result
```

Упражнение 13.9. Мини-проект. Воспроизведите звук, создаваемый пилообразной волной с частотой 441 Гц, и сравните его со звуком, получающимся при воспроизведении прямоугольной и синусоидальной волн с той же частотой.

Решение. Вот как можно определить функцию модифицированной пилообразной волны с амплитудой 8000 и частотой 441, а затем произвести выборку массива для передачи в PyGame:

```
def modified_sawtooth(t):
    return 8000 * sawtooth(441*t)
```

```
arr = sample(modified_sawtooth,0,1,44100)
sound = pygame.sndarray.make_sound(arr)
sound.play()
```

Люди часто находят сходство в звучании пилообразной волны и струнных инструментов, например скрипки.

КРАТКИЕ ИТОГИ ГЛАВЫ

- Звуковые волны — это изменения давления воздуха с течением времени, достигающие наших ушей, которые мы воспринимаем как звуки. Звуковую волну можно считать функцией, которая в общих чертах представляет изменение давления воздуха во времени.
- PyGame и большинство других цифровых аудиосистем воспроизводят звук *дискретным* способом. Вместо функции, определяющей звуковую волну, они используют массивы значений этой функции, отобранных через равные промежутки времени. Например, при производстве аудио-компакт-дисков одна секунда звучания обычно представлена 44 100 значениями.
- Звуковые волны произвольной формы звучат как шум, а волны, имеющие форму, повторяющуюся через фиксированные интервалы, звучат как четко определенные музыкальные ноты. Функция, повторяющая свои значения через определенный интервал, называется *периодической функцией*.
- Функции синуса и косинуса — это периодические функции, их графики имеют вид повторяющихся волнообразных форм и называются *синусоидами*.
- Значения синуса и косинуса повторяются через каждые 2π единиц. Это значение называется *периодом*. *Частота* периодической функции обратна периоду и для синуса и косинуса равна $1/(2\pi)$.
- Функция вида $\sin(2n\pi t)$ или $\cos(2n\pi t)$ имеет частоту n. Чем выше частота функции звуковой волны, тем более высокую ноту она производит.
- Максимальная высота периодической функции называется ее *амплитудой*. Умножение функции синуса или косинуса на число увеличивает амплитуду функции и громкость соответствующей звуковой волны.
- Чтобы создать эффект одновременного воспроизведения двух звуков, можно сложить соответствующие им функции и получить новую функцию и новую звуковую волну. В общем случае вы можете использовать любую линейную комбинацию существующих звуковых волн для создания новой звуковой волны.

- Линейная комбинация постоянной функции с функциями вида sin $(2n\pi t)$ и cos $(2n\pi t)$ для различных значений n называется *рядом Фурье*. Несмотря на то что ряды Фурье построены из гладких синусоидальных и косинусоидальных функций, они могут хорошо аппроксимировать любые периодические функции, даже с острыми углами, такими как прямоугольные волны.
- Постоянную функцию в комбинации с синусами и косинусами разных частот можно рассматривать как базис пространства периодических функций. Линейная комбинация этих базисных функций, которая наилучшим образом аппроксимирует данную функцию, называется *коэффициентами Фурье*.
- Чтобы найти компоненту заданного двух- или трехмерного вектора в направлении некоторого базисного вектора, можно использовать скалярное произведение этого вектора на вектор стандартного базиса.
- Аналогично можно вычислить специальное внутреннее произведение периодической функции на функцию синуса или косинуса, чтобы найти компоненту, связанную с этой функцией. Внутреннее произведение периодических функций — это определенный интеграл, взятый по заданному диапазону, в нашем случае от нуля до единицы.

Часть III

Машинное обучение

В части III мы используем все, что вы узнали о математических функциях, векторах и вычислениях, для реализации некоторых алгоритмов машинного обучения. Вокруг машинного обучения много шума, поэтому стоит уточнить, что это такое. Машинное обучение — это часть области создания *искусственного интеллекта*, в которой изучаются способы разработки компьютерных программ для интеллектуального решения задач. Если вам приходилось играть в видеоигры против компьютера, то знайте, что вы взаимодействовали с искусственным интеллектом. Такой противник обычно запрограммирован набором правил, которые помогают ему уничтожить, перехитрить или как-то иначе победить вас.

Чтобы алгоритм можно было классифицировать как алгоритм *машинного обучения*, он должен не только работать автономно и разумно, но и учиться на собственном опыте. Это означает, что чем больше данных он получает, тем лучше справляется с поставленной задачей. Следующие три главы посвящены особому виду машинного обучения, называемому *обучением с учителем*. Разрабатывая алгоритмы обучения с учителем, мы даем им *обучающие* наборы данных — пары входных и соответствующих им выходных данных, после чего алгоритмы должны научиться, получив новые входные данные, самостоятельно выдавать правильные выходные данные. В этом смысле результатом обучения алгоритма является новая математическая функция, которая может эффективно отображать некоторые входные данные в некоторое решение на выходе.

В главе 14 мы рассмотрим простой алгоритм обучения с учителем, называемый *линейной регрессией*, и используем его для прогнозирования цены подержанного автомобиля на основе его пробега. Набор обучающих данных для этого алгоритма состоит из известного пробега и цены многих подержанных автомобилей. Не имея никаких предварительных знаний об оценке автомобилей, наш алгоритм научится определять цену автомобиля на основе его пробега. Алгоритм линейной регрессии работает, принимая пары (x, p), где x — пробег, p — цена, и находя линейную функцию, которая наилучшим образом их аппроксимирует. Работа алгоритма сводится к поиску уравнения прямой, которая точнее соответствует всем известным точкам (x, p) в двухмерном пространстве. Большая часть нашей работы будет заключаться в выяснении значения слова «точнее»!

В главах 15 и 16 мы рассмотрим другой тип задач обучения с учителем, называемый *классификацией*. Для любой точки числовых входных данных задачи этого вида должны ответить на вопрос «да/нет» или предложить несколько вариантов ответа. В главе 15 создадим алгоритм, просматривающий данные с пробегом и ценами для двух разных моделей автомобилей и пытающийся правильно идентифицировать модель автомобиля на основе новых данных, которых прежде он не видел. И снова работа алгоритма сводится к поиску функции, которая точнее соответствует значениям в обучающем наборе данных, и мы снова должны будем решить, что означает точность для функции, отвечающей на вопрос «да» или «нет».

В главе 16 усложним задачу классификации. На этот раз набором входных данных будут служить изображения рукописных цифр от 0 до 9, а желаемым результатом — значение рукописной цифры. Как мы видели в главе 6, изображение состоит из множества данных — можно рассматривать изображения как представления в многомерных векторных пространствах. Чтобы справиться с этой сложностью, мы используем специальный тип математической функции, называемый *многослойным персептроном*. Это особый вид *искусственной нейронной сети* и один из самых обсуждаемых в настоящее время алгоритмов машинного обучения.

Возможно, вы не станете экспертом по машинному обучению, прочитав эти три короткие главы, но я надеюсь, что у вас появится прочная основа для дальнейшего изучения предмета. В частности, они должны сорвать покров таинственности с предмета машинного обучения. Мы не будем как по волшебству наполнять наши компьютеры человеческим разумом, а станем обрабатывать реальные данные, используя Python, а затем творчески применять математические знания, полученные к настоящему моменту.

14

Подгонка функций под данные

В этой главе

- ✓ Измерение близости моделирования набора данных функцией.
- ✓ Исследование пространств функций, определяемых константами.
- ✓ Использование градиентного спуска для оптимизации качества подгонки.
- ✓ Моделирование наборов данных различными видами функций.

Приемы матанализа, которые мы изучили в части II, требуют применения корректных функций. Для существования производной, как мы уже знаем, функция должна быть достаточно гладкой, а для вычисления точной производной или интеграла нужна функция, имеющая простую формулу. Но в реальном мире с реальными данными дело обстоит намного сложнее. Из-за случайностей или ошибок измерения мы редко встречаем идеально гладкие функции. В этой главе посмотрим, как можно взять беспорядочные данные и смоделировать их с помощью простой математической функции. Эта задача называется *регрессией*.

Я приведу пример использования реального набора данных, состоящего из сведений о 740 подержанных автомобилях, выставленных на продажу на веб-сайте CarGraph.com. Все они относятся к одной модели — Toyota Prius, и для каждого имеется информация о пробеге и цене продажи. Нанеся эти данные на график (рис. 14.1), можно заметить, что наблюдается явная тенденция к снижению цены с увеличением пробега. Это свидетельствует о том, что автомобили теряют

ценность по мере эксплуатации. Наша цель — придумать простую функцию, описывающую изменение цены подержанного Prius с увеличением пробега.

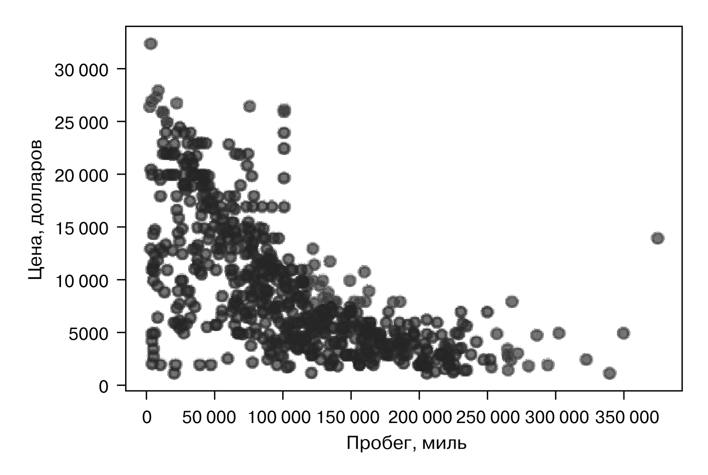


Рис. 14.1. График зависимости цены от пробега подержанных автомобилей Toyota Prius, выставленных на продажу на сайте CarGraph.com

Мы не можем нарисовать график гладкой функции, проходящий через все эти точки, а если бы и могли, то это было бы бессмысленно. Многие из этих точек являются исключениями и, вероятно, ошибочны (например, на рис. 14.1 можно видеть несколько почти новых автомобилей, которые продаются менее чем за 5000 долларов). Безусловно, есть и другие факторы, влияющие на цену продажи подержанного автомобиля. Мы не должны ожидать, что стоимость автомобиля определяется только его пробегом.

Единственное, что здесь можно сделать, — найти функцию, аппроксимирующую тренд этих данных. Функция $p(x)$ должна принимать пробег x и возвращать типичную цену Prius с данным пробегом. Для этого нужно предположить, что это будет за функция. Для начала возьмем самый простую функцию — линейную.

В главе 7 мы рассматривали разные способы представления линейных функций, здесь используем формулу вида $p(x) = ax + b$, где x — пробег автомобиля, p — его цена, а коэффициенты a и b — числа, определяющие форму функции. При выборе a и b эта функция представляет воображаемый механизм, принимающий пробег автомобиля Toyota Prius и предсказывающий его цену, как показано на рис. 14.2.

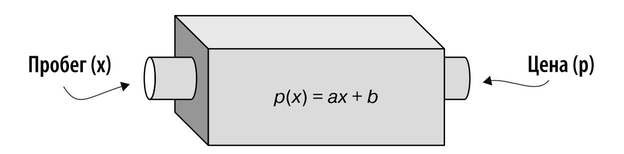


Рис. 14.2. Линейная функция, предсказывающая цену *p* по пробегу *x*

Напомню, что коэффициент a задает наклон прямой, а b — ее значение в точке с нулевой координатой x. При таких значениях, как $a = -0{,}05$ и $b = 20\,000$, график функции выглядит как прямая, начинающаяся в точке с ценой 20 000 долларов и уменьшающейся на 0,05 доллара с каждой пройденной милей (рис. 14.3).

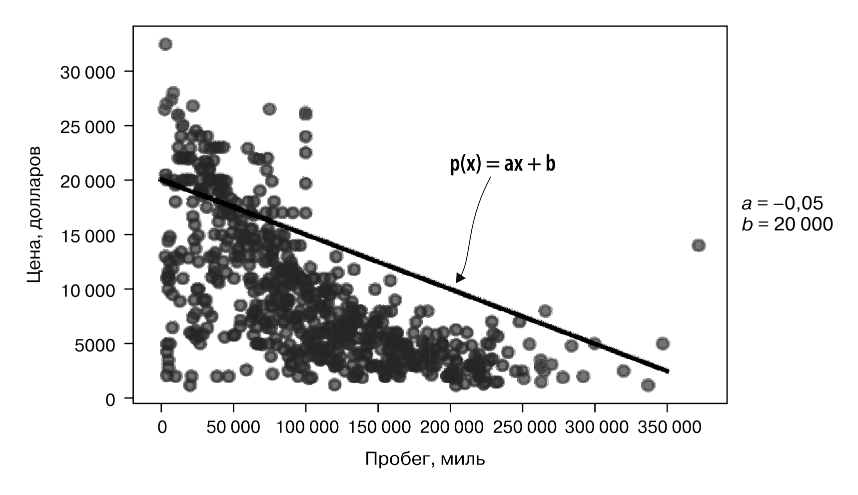


Рис. 14.3. Прогнозирование цены на Prius на основе пробега с использованием функции вида $p(x) = ax + b$, где $a = -0{,}05$, $b = 20\,000$

Этот выбор функции прогнозирования подразумевает, что новый автомобиль Prius стоит 20 000 долларов и теряет в цене 0,05 с каждой пройденной милей. Эти значения могут быть или не быть правильными, на самом деле есть все основания полагать, что они не идеальны, потому что график прямой не соответствует большинству данных. Задача поиска значений a и b таких, при которых $p(x)$ как можно точнее соответствовала бы данным, называется *линейной регрессией*. Найдя наилучшие значения коэффициентов, мы сможем сказать, что $p(x)$ — это *прямая наилучшего соответствия*.

Занимаясь приближением $p(x)$ к реальным данным, кажется разумным предположить, что наклон a прямой должен быть отрицательным, чтобы прогнозируемая цена уменьшалась с увеличением пробега. Однако нам не нужно делать таких предположений, потому что мы можем реализовать алгоритм, который вычислит этот факт непосредственно из исходных данных. Вот почему регрессия — это простой пример алгоритма машинного обучения: основываясь только на данных, он делает вывод о тенденции, а затем может делать прогнозы относительно новых точек данных.

Единственное реальное ограничение, которое мы накладываем, — наш алгоритм ищет линейные функции. *Линейная функция* предполагает, что скорость уменьшения цены постоянна и за первые 1000 миль пробега автомобиль теряет в цене столько же, сколько и за 1000 миль при общем пробеге больше 100 000 миль.

Здравый смысл говорит, что это не так и на самом деле автомобили теряют значительную часть своей цены в тот момент, когда их увозят со стоянки. Но наша цель не в том, чтобы найти идеальную модель, а в том, чтобы найти простую модель, которая работает достаточно хорошо.

Первое, что нужно сделать, — определить правила оценки: насколько хорошо линейная функция предсказывает цену автомобиля Prius по его пробегу. Для этого нужно написать функцию на Python, называемую *функцией потерь* или *функцией затрат*, которая принимает функцию $p(x)$ и возвращает число, служащее оценкой близости функции $p(x)$ к исходным данным. Затем мы сможем измерить, насколько хорошо функция $p(x) = ax + b$ соответствует набору данных для любой пары чисел a и b. Для каждой пары (a, b) существует одна линейная функция, поэтому мы можем рассматривать задачу как исследование двухмерного пространства таких пар и оценку линейных функций, которые они представляют.

На рис. 14.4 показано, что при выборе положительных значений a и b получается прямая, направленная вверх. Если бы это была искомая функция цены, то получалось бы, что цена автомобиля увеличивается с каждой пройденной милей, а это маловероятно.

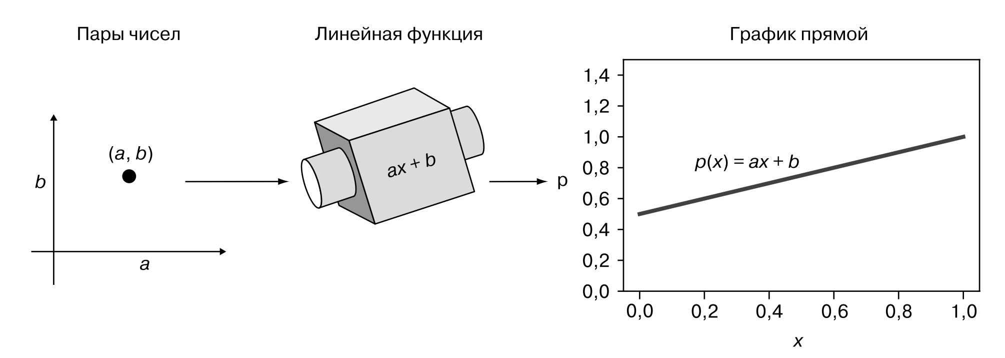


Рис. 14.4. Пара чисел (a, b) определяет линейную функцию, которую можно изобразить на графике в виде прямой линии. График функции с положительным значением a направлен вверх

Функция потерь сравнивает прямую с фактическими данными и возвращает большое число, указывающее, что прямая располагается далеко от данных. Чем ближе прямая подходит к данным, тем меньше потери и лучше соответствие.

Нам нужны такие значения a и b, которые соответствуют не просто маленькому значению функции потерь, но *наименьшему* из возможных. Вторая важная функция, которую мы напишем, называется `linear_regression`. Она будет автоматически находить наилучшие значения a и b, определяющие прямую наилучшего

соответствия. Чтобы реализовать ее, создадим функцию, сообщающую потери для любых значений a и b, и минимизируем ее, используя прием градиентного спуска, описанный в главе 12. Начнем с реализации функции потерь, чтобы получить возможность измерить, насколько хорошо какая-то функция соответствует набору данных.

14.1. ИЗМЕРЕНИЕ КАЧЕСТВА СООТВЕТСТВИЯ ФУНКЦИИ

Напишем функцию потерь так, чтобы она могла работать с любым набором данных, а не только с коллекцией подержанных автомобилей. Это позволит нам протестировать ее на более простых (созданных искусственно) наборах данных, чтобы убедиться в правильной ее работе. Итак, функция потерь — это функция на языке Python, принимающая два аргумента. Один из них — функция $f(x)$, которую требуется проверить, а второй — набор данных для проверки с парами (x, y). Для примера с подержанными автомобилями функция $f(x)$ может быть линейной функцией, дающей цену автомобиля с любым пробегом, а пары (x, y) — это фактические значения пробега и цены из набора данных.

Результат функции потерь — это одно число, оценивающее, как далеко значения $f(x)$ отклоняются от правильных значений y. Если $y = f(x)$ для каждого значения x, то это означает, что функция идеально соответствует данным и функция потерь вернет ноль. Однако более вероятно, что функция не будет точно соответствовать всем точкам данных и функция потерь будет возвращать некоторое положительное число. На самом деле мы напишем две функции потерь, чтобы сравнить их и получить представление о том, как они работают:

- `sum_error` — складывает отклонения $f(x)$ от y для каждой пары (x, y) в наборе данных;
- `sum_square_error` — складывает квадраты этих отклонений.

На практике чаще всего используется вторая функция, и вскоре вы поймете почему.

14.1.1. Измерение отклонения функции

В примерах с исходным кодом для этой книги вы найдете специально созданный набор данных `test_data`. Это список значений (x, y) на Python, в котором значения x находятся в диапазоне от -1 до 1. Я намеренно выбрал значения y так, чтобы точки лежали близко к линии $f(x) = 2x$. На рис. 14.5 вместе с прямой — графиком этой функции показаны точки из набора данных `test_data`.

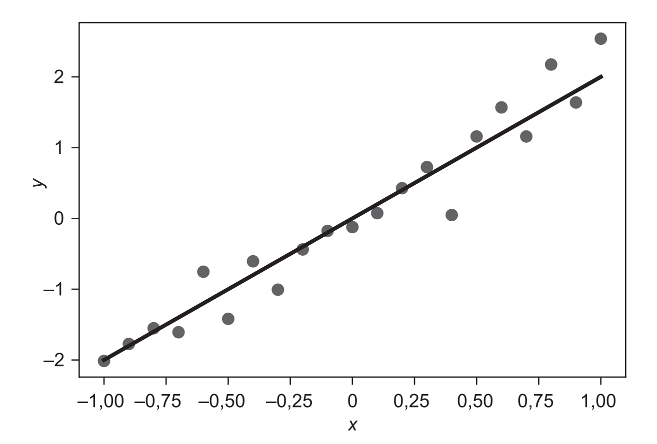


Рис. 14.5. Набор данных, сгенерированных так, чтобы они располагались примерно вдоль прямой функции $f(x) = 2x$

Близость $f(x) = 2x$ к набору данных означает, что для любого значения x в наборе данных значение $2x$ — это довольно хорошая оценка соответствующего значения y. Например, точка на прямой $(x, y) = (0{,}2, 0{,}427)$ практически совпадает с фактическим значением из набора данных. Для значения $x = 0{,}2$ функция $f(x) = 2x$ предсказала бы $y = 0{,}4$. Абсолютное значение разности $|f(0{,}2) - 0{,}4|$ сообщает нам величину ошибки, которая составляет примерно 0,027.

Значение ошибки — разность между фактическим значением y и значением, предсказанным функцией $f(x)$, — можно изобразить как расстояние по вертикали от фактической точки (x, y) до прямой — графика f. На рис. 14.6 показаны расстояния, обозначающие ошибки, изображенные как вертикальные отрезки.

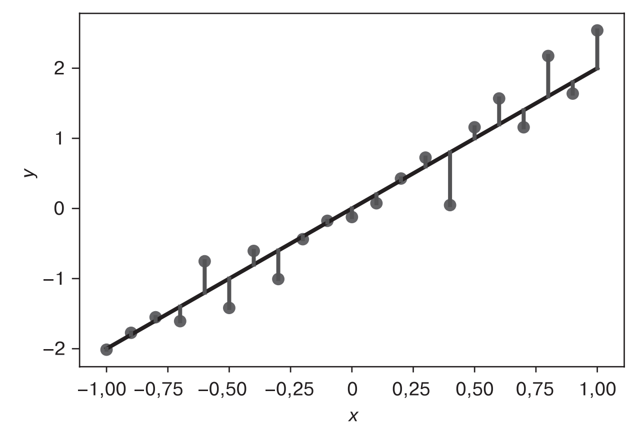


Рис. 14.6. Значения ошибок определяются как разности между функцией $f(x)$ и фактическими значениями y

Некоторые из этих ошибок меньше, другие больше, но как количественно оценить качество соответствия? Сравним этот график с графиком функции $g(x) = 1 - x$, которая явно плохо соответствует данным (рис. 14.7).

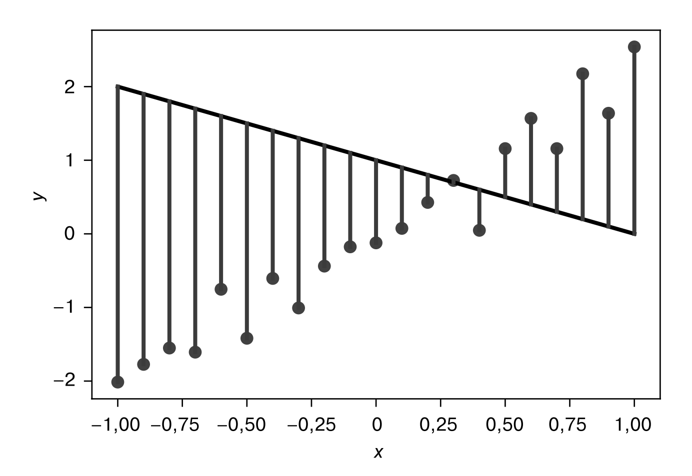


Рис. 14.7. Функция с большими значениями ошибок

Функция $g(x) = 1 - x$ близка только к одной из точек, но общая сумма ошибок (отклонений) намного больше. Соответственно, мы можем написать первую функцию потерь, которая просто складывает все ошибки. Чем больше сумма ошибок, тем хуже соответствие, а чем меньше — тем лучше. В этой функции мы просто переберем все пары (x, y), вычислим для каждой из них абсолютное значение разности между $f(x)$ и y и сложим результаты:

```
def sum_error(f,data):
    errors = [abs(f(x) - y) for (x,y) in data]
    return sum(errors)
```

Для проверки этой функции воплотим $f(x)$ и $g(x)$ в код:

```
def f(x):
    return 2*x

def g(x):
    return 1-x
```

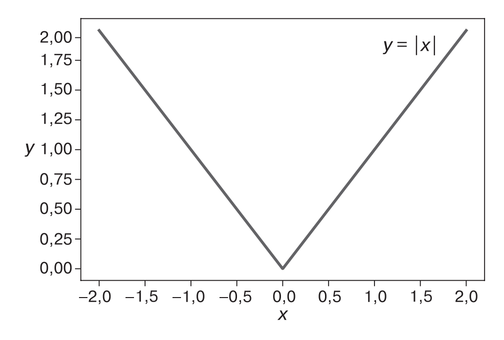
Как и ожидалось, сумма ошибок для $f(x) = 2x$ получилась меньше, чем для $g(x) = 1 - x$:

```
>>> sum_error(f,test_data)
5.021727176394801
>>> sum_error(g,test_data)
38.47711311130152
```

Точные значения полученных результатов не важны — важно их отношение «больше/меньше». Поскольку сумма ошибок для $f(x)$ меньше, чем для $g(x)$, можно утверждать, что $f(x)$ лучше соответствует заданным данным.

14.1.2. Суммирование квадратов ошибок

Функция `sum_error` — возможно, наиболее очевидный способ измерения расстояния от прямой до точки данных, но на практике мы будем использовать функцию потерь, вычисляющую сумму квадратов ошибок. Тому есть несколько веских причин. Самая простая заключается в том, что функция квадрата расстояния гладкая, поэтому для ее минимизации можно применять производные. В отличие от нее функция абсолютных значений расстояний не гладкая и поэтому не во всех точках имеет производную. Взгляните на графики функций $|x|$ и x^2 (рис. 14.8) — они обе возвращают значения тем больше, чем дальше x от 0, но только вторая гладкая в точке $x = 0$ и имеет там производную.



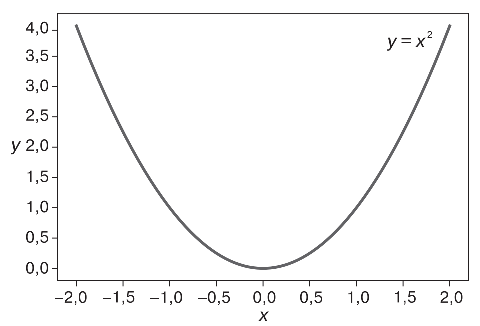


Рис. 14.8. График функции $y = |x|$ не гладкий в точке $x = 0$, а график $y = x^2$ — гладкий

Имея функцию $f(x)$ для проверки, можно просмотреть каждую пару (x, y) и сложить значения $(f(x) - y)^2$. Именно это делает функция `sum_squared_error`, и ее реализация немногим отличается от реализации `sum_error` — нужно лишь возвести ошибку в квадрат вместо взятия абсолютного значения:

```
def sum_squared_error(f,data):
    squared_errors = [(f(x) - y)**2 for (x,y) in data]
    return sum(squared_errors)
```

Эту функцию потерь тоже можно визуализировать, только вместо вертикальных отрезков, соединяющих точки с прямой — графиком функции, ошибки можно рассматривать как квадраты со стороной, равной расстоянию от точки до прямой, площадь каждого квадрата — это квадрат ошибки для точки, а общая площадь всех квадратов — результат, возвращаемый `sum_squared_error`. Общая площадь

квадратов на рис. 14.9 сообщает сумму квадратов ошибок между `test_data` и $f(x) = 2x$. (Обратите внимание на то, что квадраты на этом графике не похожи на квадраты, потому что оси x и y имеют разный масштаб!)

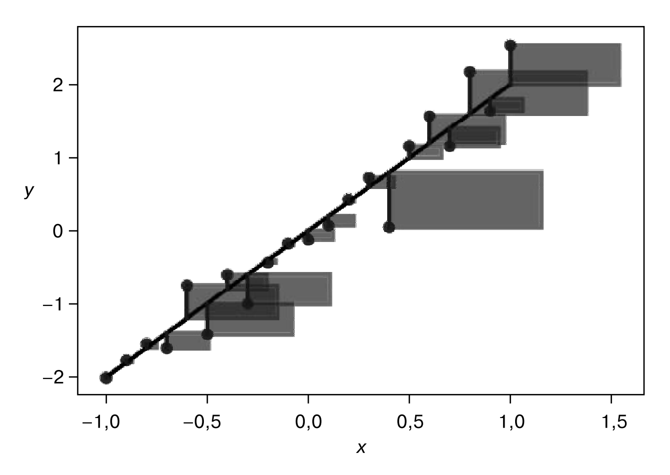


Рис. 14.9. Визуальное представление суммы квадратов ошибок между функцией и набором данных

Значение y, отстоящее в два раза дальше от графика на рис. 14.9, вносит в сумму квадратов ошибок в четыре раза больший вклад. Одна из причин предпочтительности этой функции потерь заключается в том, что она более агрессивно реагирует на плохое соответствие. Например, квадраты ошибок для $h(x) = 3x$ намного больше, как можно видеть на рис. 14.10.

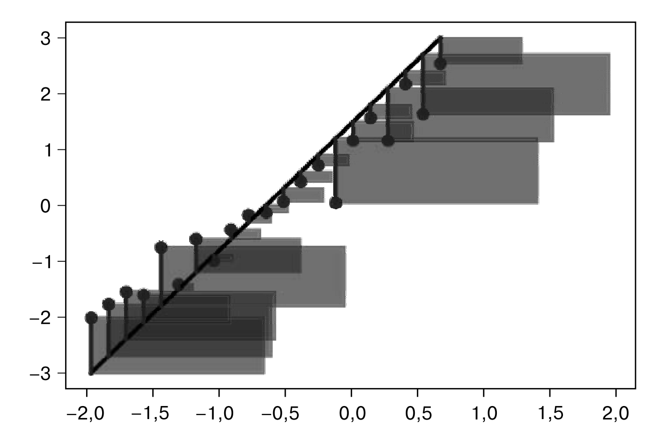


Рис. 14.10. Визуальное представление sum_squared_error для $h(x) = 3x$ относительно тестовых данных

Мы не будем пытаться изобразить квадраты ошибок для $g(x) = 1 - x$, потому что они настолько велики, что заполняют почти всю область диаграммы и значительно перекрывают друг друга. Однако можем сравнить значения, возвращаемые `sum_squared_error` для $f(x)$ и $g(x)$, и убедиться, что разница между ними еще более значительная, чем при использовании `sum_error`:

```
>>> sum_squared_error(f,test_data)
2.105175107540148
>>> sum_squared_error(g,test_data)
97.1078879283203
```

График $y = x^2$ на рис. 14.8 явно гладкий, и, как оказывается, при изменении параметров a и b линейной функции функция потерь тоже изменяется гладко. По этой причине мы продолжим использовать функцию потерь `sum_squared_error`.

14.1.3. Вычисление потерь для функций цены автомобиля

Я начну с обоснованного предположения, что автомобили Prius обесцениваются с увеличением пробега. Существует несколько разных моделей Toyota Prius, но я предположу, что средняя розничная цена составляет примерно 25 000 долларов. Чтобы упростить расчеты, в первой своей модели предположим, что максимальный пробег составляет 125 000 миль, после чего цена автомобилей становится равной 0 долларов. Это означает, что автомобили обесцениваются в среднем на 0,2 доллара за милю, то есть цена p автомобиля Prius определяется по его пробегу x путем вычитания $0{,}2x$ долларов из начальной цены 25 000 долларов, а это значит, что $p(x)$ — линейная функция, поскольку имеет знакомую форму $p(x) = ax + b$, где $a = -0{,}2$ и $b = 25\,000$:

$$p(x) = -0{,}2x + 25\,000.$$

Посмотрим, как выглядит график этой функции на фоне данных из CarGraph (рис. 14.11). Данные и код на Python для построения диаграммы вы найдете в примерах исходного кода для этой главы.

Очевидно, что многие автомобили в наборе данных преодолели установленный мною предел пробега в 125 000 миль. Это может означать, что предположение о норме уменьшения цены слишком завышено. Попробуем установить норму равной 0,1 доллара за милю, подразумевая функцию цены

$$p(x) = -0{,}1x + 25\,000.$$

Эта функция тоже не идеальна. На графике (рис. 14.12) видно, что эта функция завышает цену большинства автомобилей.

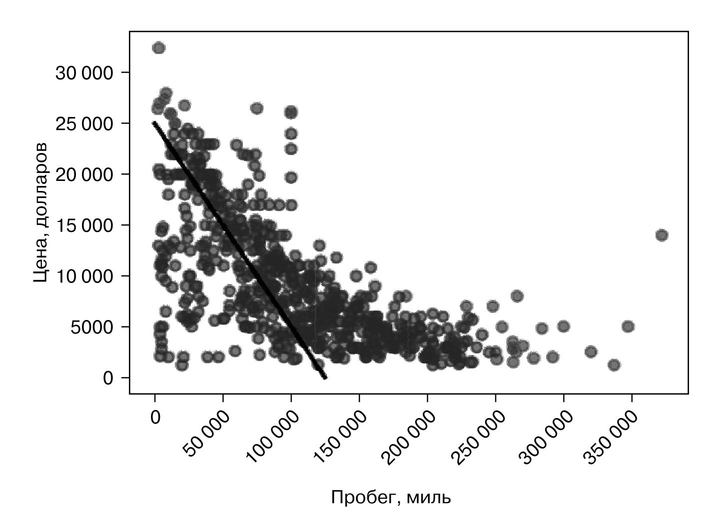


Рис. 14.11. Данные с ценами и пробегом подержанных Prius и моя гипотетическая функция цены

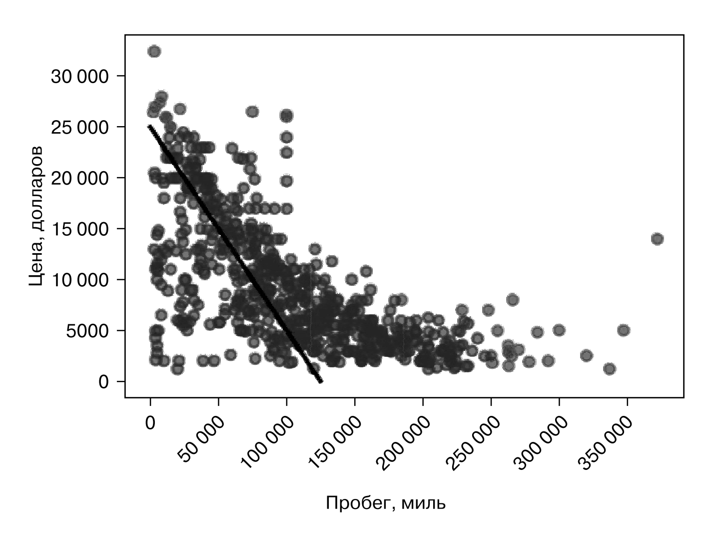


Рис. 14.12. График функции, предполагающей норму уменьшения цены 0,1 доллара за милю

Можно также поэкспериментировать с начальной ценой, которая, как мы предположили, составляет 25 000 долларов. Как ни странно, автомобиль теряет большую часть стоимости в тот момент, когда уезжает со стоянки, поэтому цена в 25 000 долларов может быть завышенной для подержанного автомобиля с очень небольшим пробегом. Если автомобиль теряет 10 % своей цены, когда уезжает со стоянки, то выбор цены 22 500 долларов при нулевом пробеге может дать более точные результаты (рис. 14.13).

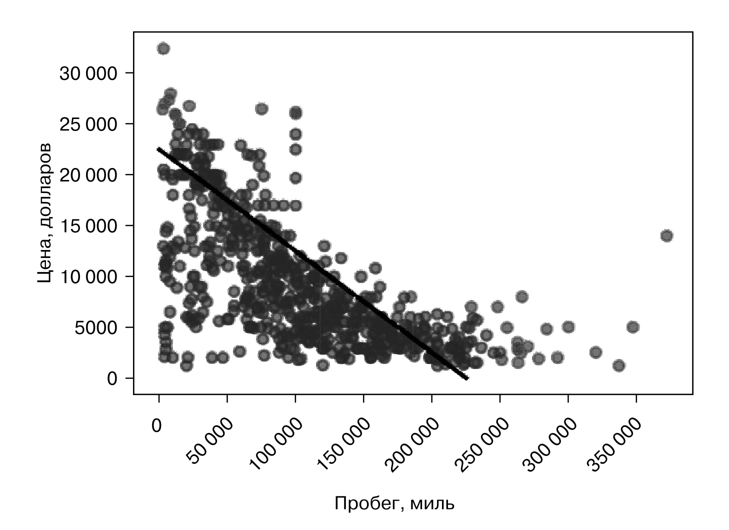


Рис. 14.13. Проверка предположения о начальной цене 22 500 долларов подержанных Toyota Prius

Мы можем потратить много времени на размышления о том, какая линейная функция лучше соответствует данным, но чтобы увидеть, улучшаются ли результаты с теми или иными предположениями, нужно использовать функцию потерь. Применив функцию `sum_squared_error`, можно оценить, какое из обоснованных предположений ближе всего к данным. Вот три функции определения цены, воплощенные в код на Python:

```
def p1(x):
    return 25000 - 0.2 * x

def p2(x):
    return 25000 - 0.1 * x

def p3(x):
    return 22500 - 0.1 * x
```

Функция `sum_squared_error` принимает функцию, а также список пар чисел, представляющих данные — в нашем случае значения пробега и цены:

```
prius_mileage_price = [(p.mileage, p.price) for p in priuses]
```

Применив функцию `sum_squared_error` к каждой из трех функций цены, можно сравнить их соответствие данным:

```
>>> sum_squared_error(p1, prius_mileage_price)
88782506640.24002
>>> sum_squared_error(p2, prius_mileage_price)
```

```
34723507681.56001
>>> sum_squared_error(p3, prius_mileage_price)
22997230681.560013
```

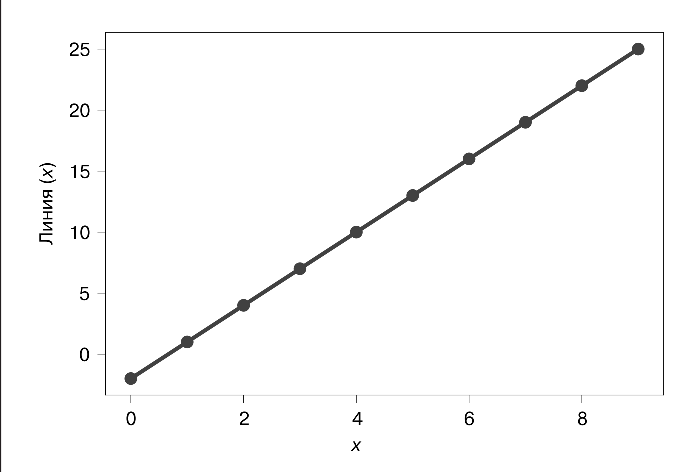
Это довольно большие значения — примерно 88,7 млрд, 34,7 млрд и 22,9 млрд соответственно. Отмечу еще раз, что сами значения не важны, важно только их соотношение «больше/меньше». Поскольку последнее значение самое маленькое, можно заключить, что p3 — лучшая из трех функций цены. Учитывая, насколько ненаучным было составление этих функций, я мог бы продолжать гадать и найти линейную функцию, которая имеет еще меньшую ошибку. Однако вместо того чтобы гадать и проверять свои догадки, предлагаю рассмотреть систематический подход к исследованию пространства возможных линейных функций.

14.1.4. Упражнения

Упражнение 14.1. Создайте набор точек данных, лежащих на прямой, и продемонстрируйте, что обе функции потерь — sum_error и sum_squared_error — возвращают ровно ноль для соответствующей линейной функции.

Решение. Вот линейная функция и некоторые точки, лежащие на ее графике:

```
def line(x):
    return 3*x-2
points = [(x,line(x)) for x in range(0,10)]
```



Обе функции, sum_error(line,points) и sum_squared_error(line,points), возвращают ноль, потому что ни одна из точек не отклоняется от линии.

Упражнение 14.2. Вычислите значение потерь для двух линейных функций, $x + 0,5$ и $2x - 1$. Какая из них дает меньшую квадратичную ошибку по сравнению с `test_data` и что это говорит о качестве соответствия?

Решение

```
>>> sum_squared_error(lambda x:2*x-1,test_data)
23.1942461283472
>>> sum_squared_error(lambda x:x+0.5,test_data)
16.607900877665685
```

Для функции $x + 0,5$ `sum_squared_error` дает меньшее значение, то есть она лучше соответствует данным `test_data`.

Упражнение 14.3. Найдите линейную функцию `p4`, которая лучше соответствует данным, чем `p1`, `p2` или `p3`. Продемонстрируйте это, показав, что потери для нее ниже, чем для `p1`, `p2` или `p3`.

Решение. Лучшее соответствие, которое мы нашли до сих пор, — это `p3` (функция $p(x) = 22\,500 - 0,1 \cdot x$). Чтобы получить функцию, еще лучше соответствующую данным, можно попробовать настроить константы в этой формуле, пока значение потерь не уменьшится. Одно наблюдение, которое вы можете сделать, заключается в том, что `p3` лучше соответствует данным, чем `p1` и `p2`, потому что мы уменьшили значение b с 25 000 до 22 500. Если еще немного уменьшить его, то соответствие станет еще лучше. Если определить новую функцию `p4` со значением $b = 20\,000$:

```
def p4(x):
    return 20000 - 0.1 * x
```

то `sum_squared_error` вернет еще меньшее значение:

```
>>> sum_squared_error(p4, prius_mileage_price)
18958453681.560005
```

Это значение меньше значений для любой из трех предыдущих функций, а значит, данная функция лучше соответствует данным.

14.2. ИССЛЕДОВАНИЕ ПРОСТРАНСТВ ФУНКЦИЙ

Мы закончили предыдущий раздел, подобрав наугад несколько функций определения цены в форме $p(x) = ax + b$, где x представляет пробег подержанного автомобиля Toyota Prius, а p — прогноз его цены. Выбирая разные значения a и b и строя график получившейся функции $p(x)$, мы могли сказать, какой выбор лучше. Функция потерь дала возможность количественно оценить соответствие функции данным, а не рассматривать их графики. Наша цель в этом разделе — систематизировать процесс оценки различных значений a и b, чтобы минимизировать функцию потерь.

Если вы выполнили последнее упражнение из раздела 14.1 и вручную нашли функцию с лучшим соответствием, то могли заметить, что сложность отчасти обусловлена необходимостью подбирать сразу *два* параметра, a и b. Как рассказывалось в главе 6, набор всех функций, таких как $p(x) = ax + b$, образует двухмерное векторное пространство. Подбирая и проверяя параметры вручную, вы слепо выбираете точки в разных направлениях в этом пространстве и надеетесь, что функция потерь уменьшится.

В этом разделе мы попытаемся понять, каков будет ландшафт двухмерного пространства возможных линейных функций, построив график функции потерь `sum_squared_error` относительно параметров a и b, определяющих линейную функцию. В частности, построим график зависимости потерь от двух параметров, a и b, которые определяют выбор $p(x)$ (рис. 14.14).

Фактическая функция, график которой мы нарисуем, принимает два числа, a и b, и возвращает одно число, отражающее величину потерь функции $p(x) = ax + b$. Назовем эту функцию `coefficient_cost(a,b)`, потому что числа a и b являются *коэффициентами*. График этой функции мы построим в виде тепловой карты подобно тому, как делали в главе 12.

В качестве разминки попробуем подобрать функцию $f(x) = ax$, лучше всего соответствующую набору данных `test_data`, который мы использовали ранее. Это более простая задача, потому что в `test_data` не так много точек данных и нужно подобрать только один параметр: $f(x) = ax$ — это линейная функция со значением b, равным нулю. График этой функции — прямая, проходящая через начало координат, а коэффициент a определяет ее наклон. Это означает, что нам нужно исследовать только одно измерение и мы можем построить график зависимости суммы квадратов ошибок от значения a, который является графиком обычной функции.

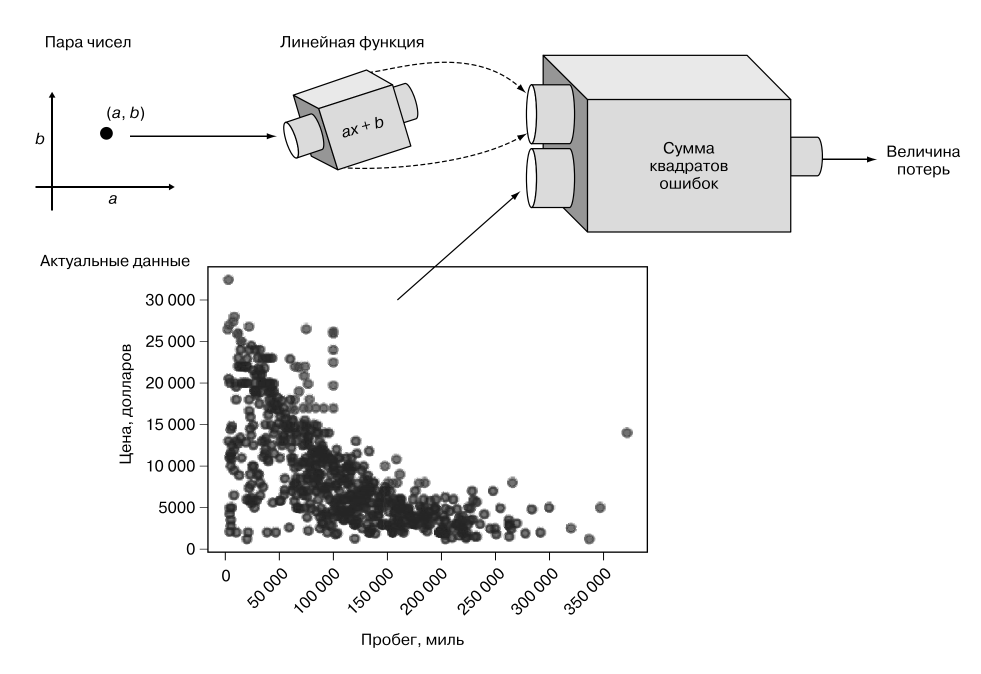


Рис. 14.14. Пара чисел (*a*, *b*) определяет линейную функцию. Сравнение ее с фактическими данными дает величину потерь в виде единственного числа

14.2.1. График функции потерь для прямых, проходящих через начало координат

Воспользуемся тем же набором данных `test_data`, что и прежде, и вычислим `sum_squared_error` для функций вида $f(x) = ax$. После этого мы сможем написать функцию `test_data_coefficient_cost`, принимающую параметр a (наклон) и возвращающую величину потерь для $f(x) = ax$. Для начала создадим функцию f аргумента a, а затем передадим ее и фактические данные в функцию потерь `sum_squared_error`:

```
def test_data_coefficient_cost(a):
    def f(x):
        return a * x
    return sum_squared_error(f,test_data)
```

Каждое значение этой функции соответствует выбранному наклону a и, следовательно, сообщает величину потерь для прямой, которую мы могли бы нарисовать поверх `test_data`. На рис. 14.15 показаны несколько значений a и соответствующие им прямые. Обратите внимание на наклон $a = -1$, который дает наибольшие потери и прямую с наихудшим соответствием.

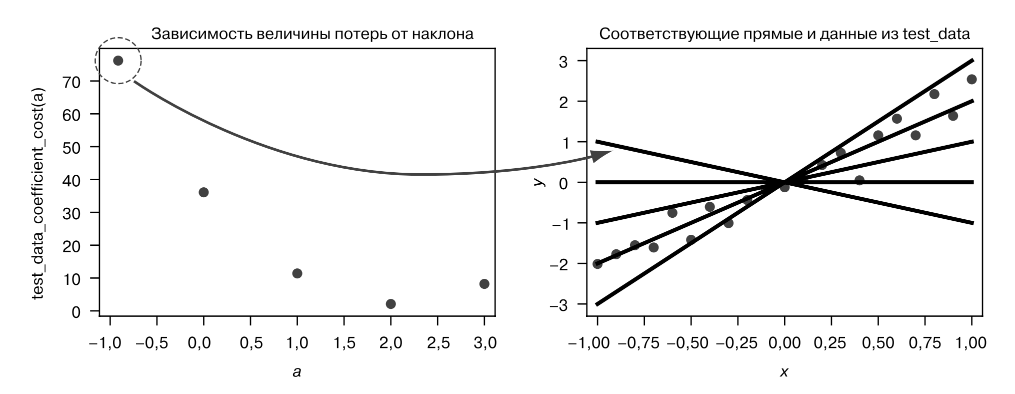


Рис. 14.15. Величина потерь для различных значений наклона *a* и соответствующих прямых

Выясняется, что функция `test_data_coefficient_cost`, построенная в диапазоне значений a, — гладкая. График на рис. 14.16 показывает, что потери уменьшаются, пока не достигают минимума около $a = 2$, а затем начинают расти.

График на рис. 14.16 сообщает, какая прямая, проходящая через начало координат, дает наименьшее значение потерь и, следовательно, наилучшее соответствие. Эта прямая имеет наклон, примерно равный 2 (точное значение мы вскоре найдем). Чтобы найти линейную функцию, лучше всего соответствующую данным о подержанных автомобилях, рассмотрим график потерь в пространстве, добавив еще одно измерение.

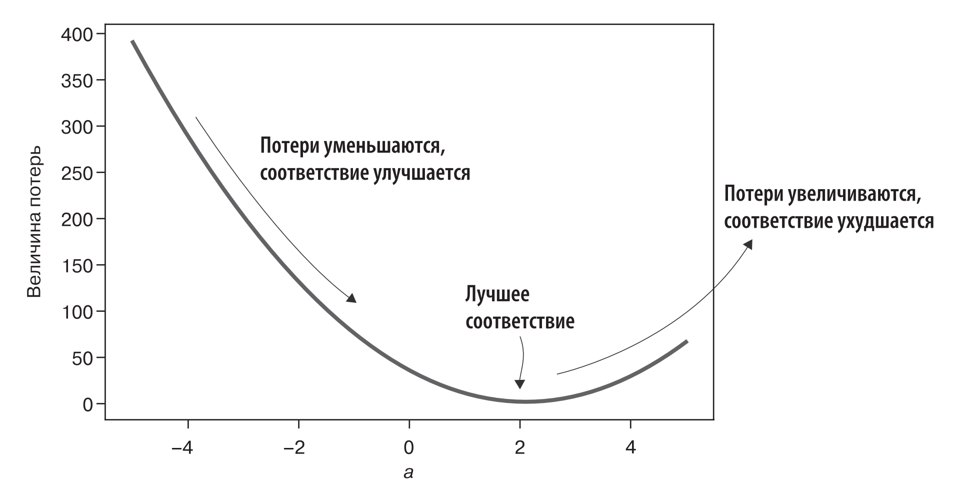


Рис. 14.16. График зависимости цены от наклона *a*, показывающий качество соответствия прямых с разным наклоном

14.2.2. Пространство всех линейных функций

Мы ищем функцию $p(x) = ax + b$, которая наиболее точно предсказывает цену автомобиля Prius, исходя из его пробега. Точность оценивается с помощью функции `sum_squared_error`. Чтобы оценить различные комбинации коэффициентов a и b, нужно сначала написать функцию `coefficient_cost(a,b)`, которая дает сумму квадратов ошибок для $p(x) = ax + b$ относительно фактических данных. Она похожа на функцию `test_data_coefficient_cost`, за исключением того, что имеет два параметра и использует другой набор данных:

```
def coefficient_cost(a,b):
    def p(x):
        return a * x + b
    return sum_squared_error(p,prius_mileage_price)
```

Теперь у нас есть двухмерное пространство пар коэффициентов (a, b), каждая из которых дает уникальную функцию $p(x)$ для сравнения с фактическими данными. На рис. 14.17 показаны две точки на плоскости ab и соответствующие прямые на графике.

Для каждой пары (a, b) и соответствующей функции $p(x) = ax + b$ можно вычислить функцию `sum_squared_error`. Именно это и делает функция `coefficient_cost` за один присест. В результате мы получаем значения потерь для всех точек на плоскости ab, которые можно изобразить в виде тепловой карты (рис. 14.18).

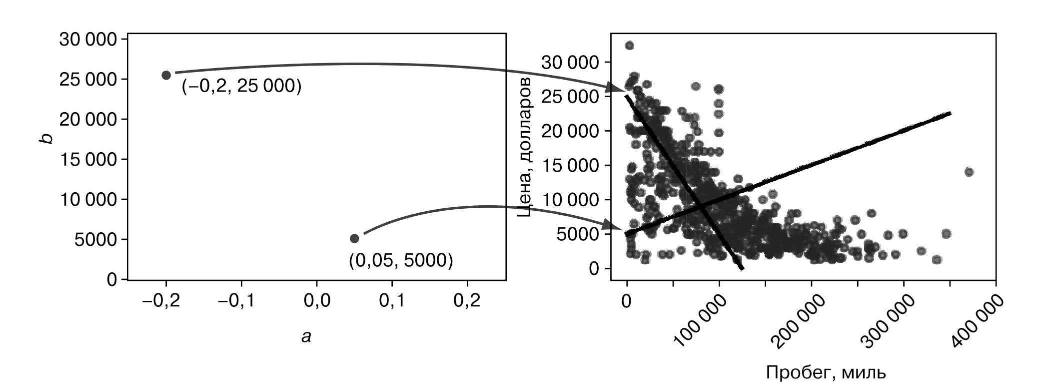


Рис. 14.17. Разные пары чисел (a, b) соответствуют разным функциям цены

На этой тепловой карте видно, что для граничных значений (a, b) функция потерь дает наиболее высокие величины. Тепловая карта темнее всего посередине, но визуально неясно, есть ли минимальное значение потерь и где именно оно находится. К счастью, у нас есть способ найти, где на плоскости (a, b) находится минимум функции потерь, — градиентный спуск.

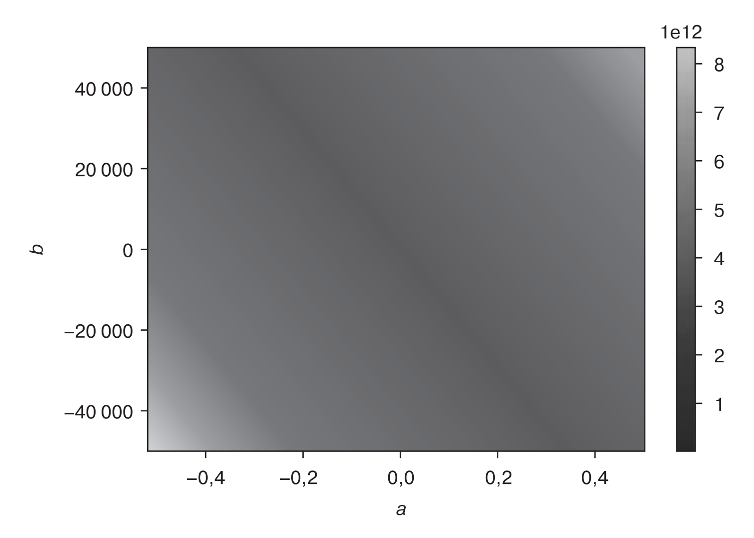


Рис. 14.18. Потери для линейных функций в виде тепловой карты значений *a* и *b*

14.2.3. Упражнения

Упражнение 14.4. Найдите точную формулу прямой, проходящей через начало координат и точку (3, 4). Сделайте это, отыскав функцию $f(x) = ax$, которая минимизирует сумму квадратов ошибок относительно этого набора данных с одной точкой.

Решение. Нужно найти один коэффициент a. Сумма квадратов ошибок говорит о том, что это квадрат разности между $f(3) = a \cdot 3$ и 4, то есть выражение $(3a - 4)^2$, которое разворачивается в $9a^2 - 24a + 16$. Это выражение можно интерпретировать как функцию потерь относительно a, то есть $c(a) = 9a^2 - 24a + 16$.

Наилучшее значение a — то, которое минимизирует потери. В этой точке производная функции потерь равна нулю. Используя правила производных из главы 10, находим $c'(a) = 18a - 24$. Это уравнение имеет решение при $a = 4/3$, то есть искомая прямая наилучшего соответствия

$$f(x) = \frac{4}{3}x.$$

Она явно проходит через начало координат и точку (4, 3).

Упражнение 14.5. Предположим, что мы используем линейную функцию для моделирования цены спортивного автомобиля в зависимости от пробега с коэффициентами $(a, b) = (-0{,}4, 80000)$. Что эта модель говорит о снижении цены автомобиля с увеличением пробега?

Решение. Значение $ax + b$ при $x = 0$ равно $b = 80\,000$. Это означает, что при нулевом пробеге автомобиль будет продан за 80 000 долларов. Значение $a = -0{,}4$ говорит о том, что значение функции $ax + b$ уменьшается со скоростью 0,4 единицы на каждую единицу увеличения x. То есть цена автомобиля уменьшается в среднем на 40 центов с каждой пройденной милей.

14.3. ПОИСК ПРЯМОЙ НАИЛУЧШЕГО СООТВЕТСТВИЯ С ПОМОЩЬЮ ГРАДИЕНТНОГО СПУСКА

В главе 12 мы использовали алгоритм градиентного спуска для минимизации гладкой функции вида $f(x, y)$. Проще говоря, находили такие значения x и y, при которых значение $f(x, y)$ было минимальным. Поскольку у нас уже реализована функция `gradient_descent`, можем просто передать ей функцию на Python, которую нужно минимизировать, и она автоматически найдет входные данные, при которых та достигает минимума.

Теперь нужно найти значения a и b, минимизирующие потери для $p(x) = ax + b$, другими словами, минимизирующие функцию `coefficient_cost(a,b)`. Передав `coefficient_cost` в функцию `gradient_descent`, мы получим пару (a, b), такую, что $p(x) = ax + b$ будет прямой наилучшего соответствия. Можно использовать найденные значения a и b, чтобы нарисовать график $ax + b$ и визуально подтвердить, что она хорошо соответствует данным.

14.3.1. Изменение масштаба данных

Есть еще одна хитрость, с которой нужно разобраться, прежде чем применять градиентный спуск. Числа, с которыми мы работали, имеют совершенно разный масштаб: норма уменьшения цены находится в диапазоне от 0 до –1, цена исчисляется десятками тысяч, а потери — сотнями миллиардов. Если не указано иное, аппроксимация производной берется с использованием значения `dx`, равного 10^{-6}. Поскольку числа так сильно различаются по величине, то при попытке применить градиентный спуск с имеющимися данными можно столкнуться с проблемой погрешности вычислений.

ПРИМЕЧАНИЕ

Я не буду вдаваться в подробности проблемы погрешности вычислений, потому что моя цель — показать, как применять математические идеи, а не как писать надежный вычислительный код. Поэтому просто покажу, как обойти эту проблему, изменив форму используемых данных.

Основываясь на нашем понимании набора данных, можно определить некоторые консервативные границы значений a и b, дающих прямую наилучшего соответствия. Значение a представляет норму уменьшения цены, поэтому наилучшее значение, вероятно, больше 0,5, то есть больше 50 центов за милю. Значение b представляет цену Prius с нулевым пробегом и определенно должно быть меньше 50 000 долларов.

Если мы определим новые переменные c и d как $a = 0{,}5 \cdot c$ и $b = 50\,000 \cdot d$, то для значений c и d меньше единицы переменные a и b должны иметь значения меньше 0,5 и 50 000 соответственно. Для a и b, меньших этих значений, функция потерь не превышает 10^{13}. Если разделить результат функции потерь на 10^{13} и выразить его через c и d, то мы получим новую версию функции потерь, аргументы и результат которой будут иметь абсолютные значения между нулем и единицей:

```
def scaled_cost_function(c,d):
    return coefficient_cost(0.5*c,50000*d)/1e13
```

Если мы найдем значения c и d, минимизирующие масштабированную функцию потерь, то сможем найти значения a и b, минимизирующие исходную функцию, используя тот факт, что $a = 0{,}5 \cdot c$ и $b = 50\,000 \cdot d$.

Это немного нестандартный подход, и существуют другие, более обоснованные способы масштабирования данных, чтобы сделать их более управляемыми в числовом выражении, один из которых рассмотрим в главе 15. Если вы хотите узнать больше, то в литературе по машинному обучению этот процесс называется *масштабированием признаков*. Теперь у нас есть все, что нужно, в том числе функция, которую можно передать алгоритму градиентного спуска.

14.3.2. Поиск и построение линии наилучшего соответствия

Функция, которую мы собираемся оптимизировать, называется `scaled_cost_function`, и можно ожидать, что ее минимум находится в точке (c, d), где $|c| < 1$ и $|d| < 1$. Поскольку оптимальные значения c и d находятся довольно близко к началу координат, мы можем начать градиентный спуск с точки $(0, 0)$. Следующий код находит искомый минимум, хотя ему может потребоваться некоторое время для этого в зависимости от быстродействия используемого компьютера:

```
c,d = gradient_descent(scaled_cost_function,0,0)
```

Этот код возвращает следующие значения *c* и *d*:

```
>>> (c,d)
(-0.12111901781176426, 0.31495422888049895)
```

Чтобы восстановить значения *a* и *b*, нужно умножить *c* и *d* на соответствующие масштабные множители:

```
>>> a = 0.5*c
>>> b = 50000*d
>>> (a,b)
(-0.06055950890588213, 15747.711444024948)
```

Наконец-то у нас есть коэффициенты, которые мы так долго искали! Округлив, можно сказать, что функция цены имеет вид

$$p(x) = -0{,}0606 \cdot x + 15\,700.$$

Это линейная функция, которая, как предполагается, имеет минимальную сумму квадратов ошибок по всему набору данных об автомобилях. Согласно найденным коэффициентам цена Toyota Prius с нулевым пробегом составляет в среднем 15 700 долларов и снижается в среднем чуть более чем на 6 центов с каждой пройденной милей. На рис. 14.19 показано, как выглядит эта прямая на фоне данных.

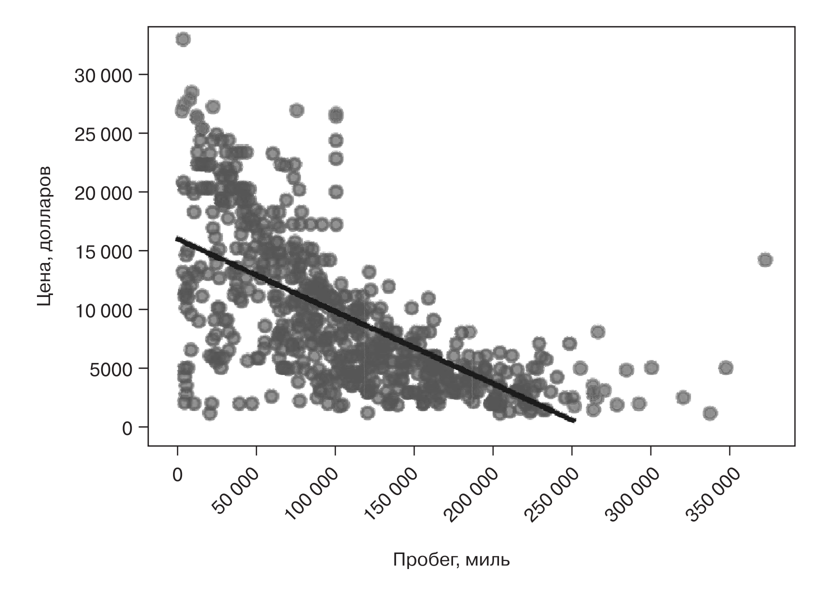


Рис. 14.19. Линия наилучшего соответствия данным о ценах на автомобили

Она выглядит, если и не лучше, то по крайней мере не хуже линейных функций $p_1(x)$, $p_2(x)$ и $p_3(x)$, которые мы подбирали. Мы можем быть уверены, что она лучше соответствует данным оценки функции потерь:

```
>>> coefficient_cost(a,b)
14536218169.403479
```

Автоматически отыскав прямую наилучшего соответствия, минимизирующую функцию потерь, мы можем сказать, что наш алгоритм научился оценивать автомобили Prius по величине их пробега и мы достигли основной цели этой главы.

Существует несколько способов вычисления линейной регрессии для получения прямой наилучшего соответствия, включая задействование некоторых оптимизированных библиотек для Python. Но независимо от методологии все они должны привести вас к одной и той же линейной функции, которая минимизирует сумму квадратов ошибок. Я выбрал методологию с применением градиентного спуска, потому что это позволило реализовать ряд идей, которые мы рассмотрели в частях I и II, а также потому что она легко обобщается. В последнем разделе главы я покажу еще одно применение градиентного спуска для регрессии, кроме того, мы будем использовать градиентный спуск и регрессию в следующих двух главах.

14.3.3. Упражнения

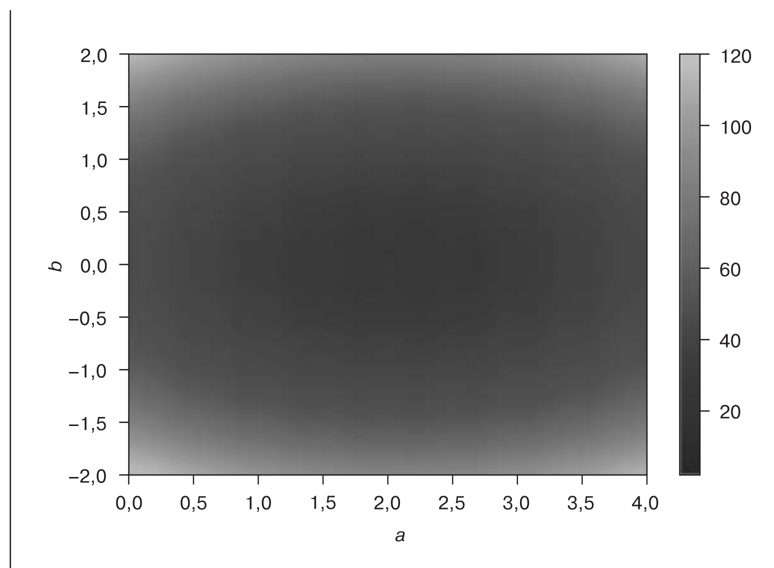
Упражнение 14.6. С помощью градиентного спуска найдите линейную функцию, которая лучше всего соответствует тестовым данным. Результирующая функция должна быть близка к $2x + 0$, но не точно совпадать с ней, потому что данные были сгенерированы случайным образом вокруг этой прямой.

Решение. Во-первых, нужно написать функцию, которая вычисляет потери $f(x) = ax + b$ относительно тестовых данных с точки зрения коэффициентов a и b:

```
def test_data_linear_cost(a,b):
    def f(x):
        return a*x+b
    return sum_squared_error(f,test_data)
```

Значения a и b, минимизирующие эту функцию, дают линейную функцию наилучшего соответствия. Ожидается, что a и b будут близки к значениям 2 и 0 соответственно, поэтому можно построить тепловую карту, чтобы понять минимизируемую функцию:

```
scalar_field_heatmap(test_data_linear_cost,-0,4,-2,2)
```



Потери $ax + b$ на тестовых данных в виде функции a и b

Похоже, что минимум этой функции потерь находится вблизи $(a, b) = (2, 0)$, как и ожидалось. Используя градиентный спуск, можно найти точные значения:

```
>>> gradient_descent(test_data_linear_cost,1,1)
(2.103718204728344, 0.0021207385859157535)
```

Этот результат означает, что прямая наилучшего соответствия определяется формулой приблизительно $2{,}10372 \cdot x + 0{,}00212$.

14.4. ПОДБОР НЕЛИНЕЙНОЙ ФУНКЦИИ

В проделанной работе отсутствовало условие, требующее, чтобы функция цены $p(x)$ была линейной. Мы выбрали линейную функцию из-за ее простоты, но тот же метод можно применить к любой функции одной переменной, определяемой двумя константами. В качестве примера найдем экспоненциальную функцию наилучшего соответствия, имеющую форму $p(x) = qe^{rx}$ и минимизирующую сумму квадратов ошибок относительно данных об автомобилях. В этом уравнении e — специальная константа 2,71828..., и мы найдем значения q и r, дающие наилучшее соответствие.

14.4.1. Особенности поведения экспоненциальных функций

Кто-то из вас, вероятно, давно не использовал экспоненциальные функции, поэтому кратко рассмотрим их особенности. Экспоненциальную функцию $f(x)$ можно распознать по аргументу x, который находится в показателе степени. Например, $f(x) = 2^x$ — это экспоненциальная функция, а $f(x) = x^2$ — нет. На самом деле $f(x) = 2^x$ — одна из самых известных экспоненциальных функций. Значение 2^x для каждого целого числа x равно числу 2, умноженному на себя x раз. В табл. 14.1 приводятся некоторые значения 2^x.

Таблица 14.1. Значения хорошо известной экспоненциальной функции 2^x

x	0	1	2	3	4	5	6	7	8	9
2*x*	1	2	4	8	16	32	64	128	256	512

Число, возводимое в степень x, называется *основанием*, поэтому в 2^x основание равно 2. Если основание больше единицы, функция возрастает с увеличением x, а если меньше единицы, то убывает. Например, для $(1/2)^x$ каждое следующее значение функции вдвое меньше предыдущего, как показано в табл. 14.2.

Таблица 14.2. Значения убывающей экспоненциальной функции $(1/2)^x$

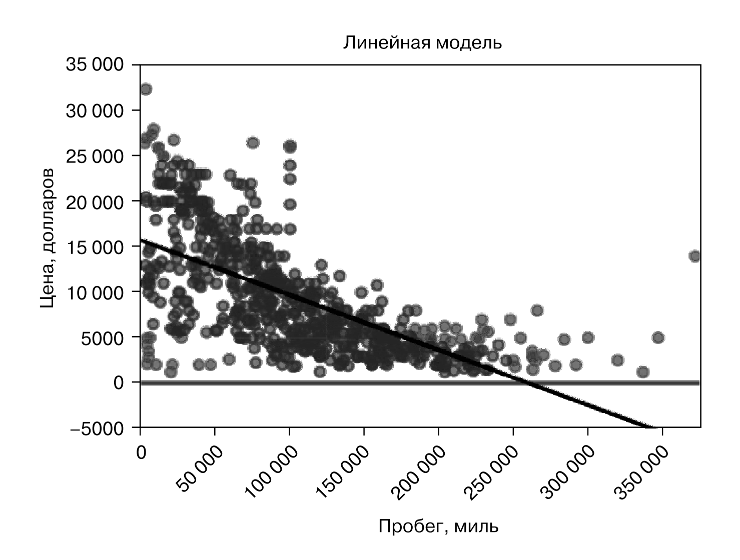
x	0	1	2	3	4	5	6	7	8	9
$(1/2)^x$	1	0,5	0,25	0,125	~0,06	~0,03	~0,015	~0,008	~0,004	~0,002

Такие функции называются *экспоненциальным спадом* или *экспоненциальным затуханием* и больше похожи на нашу модель уменьшения цены автомобиля. Экспоненциальное затухание означает, что значение функции уменьшается на одно и то же отношение на каждом интервале x фиксированного размера. Такая модель может сказать, например, что Prius теряет половину своей цены каждые 50 000 миль в том смысле, что он стоит 1/4 своей первоначальной цены после 100 000 миль пробега, и т. д.

Очевидно, что эта функция может оказаться более удачной моделью уменьшения цены. «Тойоты» — надежные и долговечные автомобили и сохраняют некоторую ценность, пока на них можно ездить. Для сравнения: наша линейная модель предполагает, что их ценность становится отрицательной по достижении определенного пробега (рис. 14.20).

Экспоненциальная функция, которую мы можем использовать, имеет вид $p(x) = qe^{rx}$, где $e = 2{,}71828...$ — фиксированное основание, а r и q — настраиваемые

коэффициенты. (Основание e может показаться выбранным произвольно или даже неудобным, тем не менее e^x — это стандартная экспоненциальная функция, поэтому к ней стоит привыкнуть.) В случае экспоненциального убывания r имеет отрицательное значение. Поскольку $e^{r \cdot 0} = e^0 = 1$, мы имеем $p(0) = qe^{r \cdot 0} = q$, поэтому q по-прежнему моделирует цену Prius с нулевым пробегом. Константа r определяет норму уменьшения цены.



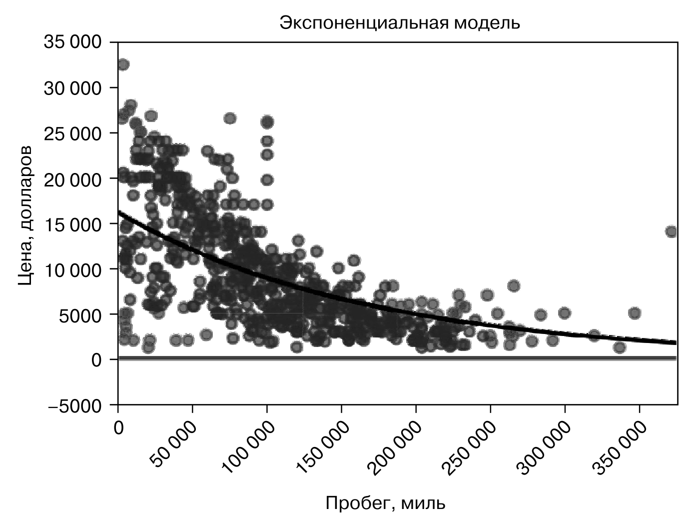


Рис. 14.20. Линейная модель предсказывает отрицательную цену Prius в отличие от экспоненциальной модели, которая показывает положительную цену при любом пробеге

14.4.2. Нахождение экспоненциальной функции наилучшего соответствия

Взяв за основу формулу $p(x) = qe^{rx}$, мы можем использовать методологию из предыдущих разделов и найти экспоненциальную функцию наилучшего соответствия. Первый шаг — написать функцию, которая принимает коэффициенты q и r и возвращает величину потерь для соответствующей функции:

```
def exp_coefficient_cost(q,r):
    def f(x):
        return q*exp(r*x)
    return sum_squared_error(f,prius_mileage_price)
```

Функция exp в языке Python вычисляет экспоненциальную функцию e_x

На следующем шаге нужно выбрать разумный диапазон для коэффициентов q и r, задающих начальную цену и норму уменьшения цены соответственно. Для q мы ожидаем, что его величина будет близка к величине b из нашей линейной модели, потому что оба коэффициента, q и b, представляют цену автомобиля с нулевым пробегом. Я буду использовать диапазон от 0 до 30 000 долларов для большей безопасности.

Оценить значение r, которое управляет нормой уменьшения цены, и установить ограничение для него немного сложнее. Уравнение $p(x) = qe^{rx}$ с отрицательным значением r подразумевает, что каждый раз, когда x увеличивается на $-1/r$ единиц, цена уменьшается в e раз, то есть она умножается на $1/e$, или примерно на 0,36. (В конце раздела я добавил упражнение, чтобы помочь вам убедиться в этом!)

На всякий случай предположим, что цена автомобиля снижается с коэффициентом $1/e$, или на 36 % от его первоначальной цены, самое раннее через 10 000 миль. Это дает $r = 10^{-4}$. Меньшее значение r будет означать более медленное уменьшение цены. Эти контрольные величины показывают нам, как изменить масштаб, и если мы разделим потери на 10^{11}, то они также останутся небольшими. Вот реализация масштабированной функции потерь, а на рис. 14.21 показана тепловая карта ее результатов:

```
def scaled_exp_coefficient_cost(s,t):
    return exp_coefficient_cost(30000*s,1e-4*t) / 1e11

scalar_field_heatmap(scaled_exp_coefficient_cost,0,1,-1,0)
```

Темная область в верхней части тепловой карты на рис. 14.21 показывает, что наименьшие потери дают малые значения t и значения s примерно в середине диапазона от 0 до 1. Мы готовы передать масштабированную функцию потерь алгоритму градиентного спуска. Результатами функции градиентного спуска являются значения s и t, минимизирующие функцию потерь, и можно выполнить обратное масштабирование, чтобы получить q и r.

```
>>> s,t = gradient_descent(scaled_exp_coefficient_cost,0,0)
>>> (s,t)
(0.6235404892859356, -0.07686877731125034)
>>> q,r = 30000*s,1e-4*t
>>> (q,r)
(18706.214678578068, -7.686877731125035e-06)
```

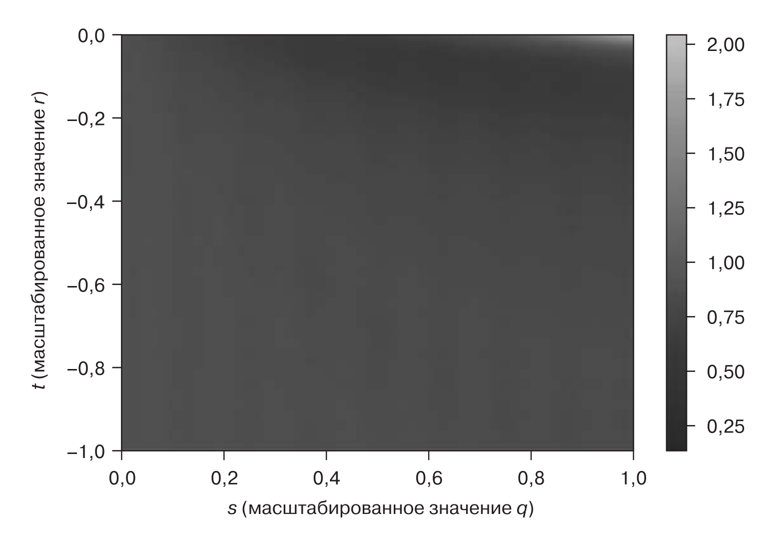


Рис. 14.21. Потери как функция масштабированных значений *q* и *r*, обозначенных *s* и *t* соответственно

Эти результаты означают, что экспоненциальная функция, которая лучше всего предсказывает цену Prius с учетом пробега, имеет вид

$$p(x) = 18\,700 \cdot e^{-0{,}00000768x}.$$

На рис. 14.22 показан график этой функции на фоне фактических данных.

Мы можем утверждать, что эта функция даже лучше линейной модели, потому что имеет меньшую сумму квадратов ошибок, то есть она лучше (пусть и немного) соответствует данным согласно функции потерь:

```
>>> exp_coefficient_cost(q,r)
14071654468.28084
```

Использование нелинейной функции, такой как экспоненциальная, — лишь один из многих вариантов метода регрессии. Мы могли бы задействовать другие нелинейные функции, например, определяемые более чем двумя константами, или данные с бóльшим числом измерений. В следующих двух главах продолжим применять функции потерь для оценки качества регрессионных моделей, а затем воспользуемся градиентным спуском, чтобы максимально улучшить соответствие.

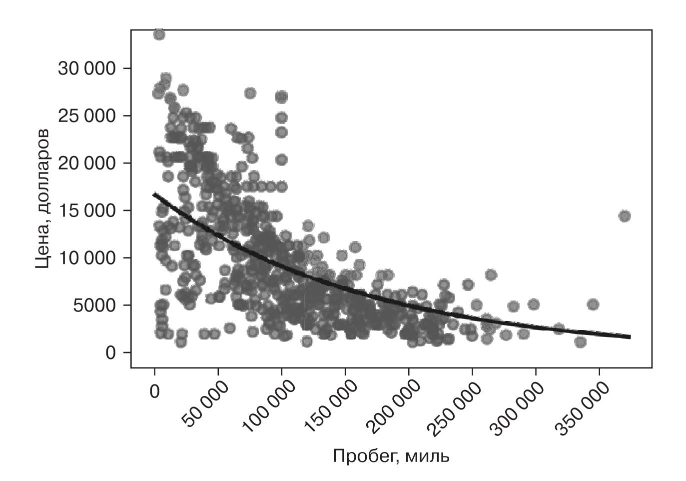


Рис. 14.22. Экспоненциальная функция наилучшего соответствия данным о ценах на автомобили Prius с пробегом

14.4.3. Упражнения

Упражнение 14.7. Убедитесь, выбрав произвольное значение r, что e^{-rx} уменьшается в e раз каждый раз, когда x увеличивается на $1/r$ единиц.

Решение. Возьмем $r = 3$, соответственно, наша тестовая функция примет вид e^{-3x}. Мы должны подтвердить, что значение этой функции уменьшается в e раз каждый раз, когда x увеличивается на 1/3 единицы. Определим функцию на Python:

```
def test(x):
    return exp(-3*x)
```

Как показано далее, при $x = 0$ функция имеет значение 1 и уменьшается в e раз при каждом увеличении x на 1/3:

```
>>> test(0)
1.0
>>> from math import e
>>> test(1/3), test(0)/e
(0.36787944117144233, 0.36787944117144233)
>>> test(2/3), test(1/3)/e
(0.1353352832366127, 0.1353352832366127)
>>> test(1), test(2/3)/e
(0.049787068367863944, 0.04978706836786395)
```

В каждом из этих случаев прибавление 1/3 к аргументу функции `test` дает тот же результат, что и деление предыдущего результата на e.

Упражнение 14.8. Какой процент цены теряют автомобили Prius через каждые 10 000 миль согласно экспоненциальной функции наилучшего соответствия?

Решение. Функция цены имеет вид $p(x) = 18\,700 \cdot e^{-0{,}00000768x}$, где значение $q = 18\,700$ представляет начальную цену в долларах. Сосредоточимся на члене $e^{rx} = e^{-0{,}00000768x}$ и посмотрим, насколько он изменится при увеличении пробега на 10 000 миль. Для $x = 0$ значение этого выражения равно 1, а для $x = 10\,000$ составляет

```
>>> exp(r * 10000)
0.9422186306357088
```

Это означает, что Prius с пробегом 10 000 миль стоит 94,2 % от первоначальной цены, то есть на 5,8 % меньше. Учитывая поведение экспоненциальной функции, это уменьшение (в процентном отношении) сохранится при увеличении любого пробега на 10 000 миль.

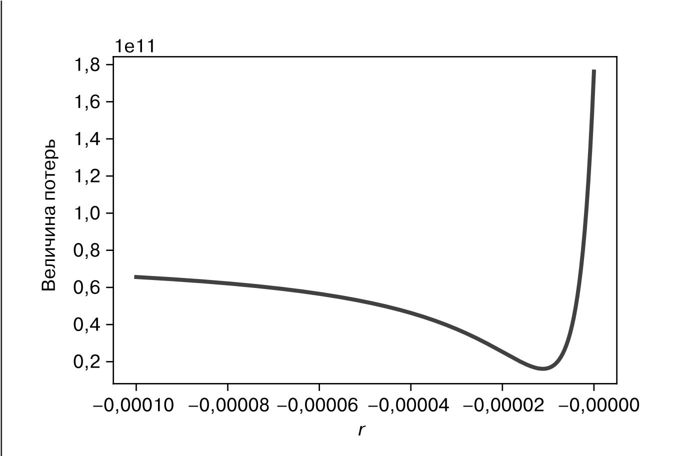
Упражнение 14.9. Пусть розничная цена автомобиля (цена с нулевым пробегом) составляет 25 000 долларов. Найдите экспоненциальную функцию, которая лучше всего соответствует данным при соблюдении этого условия. Иначе говоря, зафиксируйте $q = 25\,000$ и найдите, при каком значении r функция qe^{rx} лучше всего соответствует фактическим данным.

Решение. Мы можем написать отдельную функцию, которая дает потери экспоненциальной функции с точки зрения одного неизвестного коэффициента r:

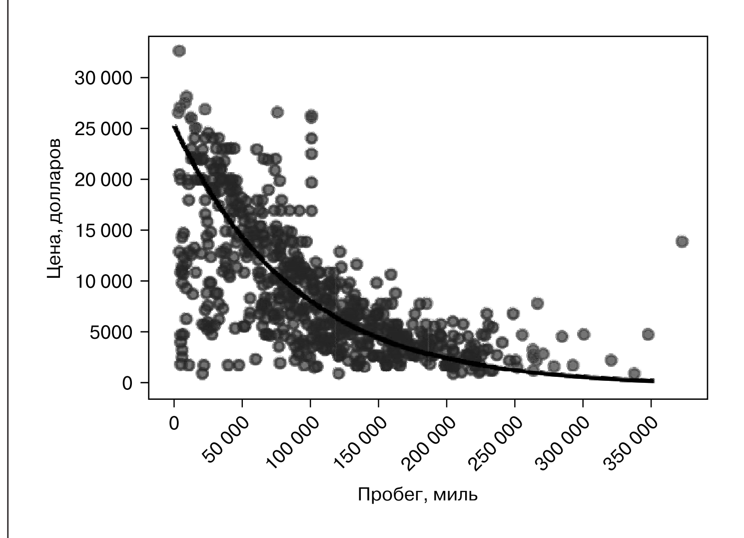
```
def exponential_cost2(r):
    def f(x):
        return 25000 * exp(r*x)
    return sum_squared_error(f,prius_mileage_price)
```

Следующий график подтверждает, что существует значение r между -10^{-4} и 0, которое минимизирует функцию потерь:

```
plot_function(exponential_cost2,-1e-4,0)
```



Похоже, что минимальное значение функция потерь имеет примерно при $r = -10^{-5}$. Чтобы автоматически минимизировать ее, нужно написать одномерную версию градиентного спуска или использовать другой алгоритм минимизации. Если хотите, можете попробовать этот подход, но поскольку параметр всего один, можно просто предположить и проверить, что $r = -1{,}12 \cdot 10^{-5}$ приблизительно соответствует значению r, дающему минимальную величину потерь. Это означает, что функция наилучшего соответствия имеет вид $p(x) = 25\,000 \cdot e^{-0{,}0000112x}$. Вот график новой экспоненциальной функции, построенный на основе исходных данных.



КРАТКИЕ ИТОГИ ГЛАВЫ

- *Регрессия* — это процесс поиска модели, описывающей отношения между различными наборами данных. В этой главе мы использовали линейную регрессию для аппроксимации цены автомобиля по его пробегу в виде линейной функции.
- Для имеющегося множества точек данных (x, y) может не существовать прямой, проходящей через все точки.
- Можно измерить, насколько близко функция $f(x, y)$, моделирующая данные, соответствует данным, вычислив расстояния между $f(x)$ и y для заданных точек (x, y).
- Функция, оценивающая, насколько хорошо модель соответствует набору данных, называется *функцией потерь*. Обычно в ее роли используется сумма квадратов расстояний от точек (x, y) до соответствующих значений модели $f(x)$. Функция, которая лучше всего соответствует данным, имеет минимальное значение функции потерь.
- В случае с линейными функциями вида $f(x) = ax + b$ каждая пара коэффициентов (a, b) определяет уникальную линейную функцию. Существует двухмерное пространство таких пар и, следовательно, двухмерное пространство прямых.
- Функция, которая принимает пару коэффициентов (a, b) и вычисляет величину потерь $ax + b$, фактически является функцией, принимающей двухмерную точку и возвращающей число. Минимизация этой функции дает коэффициенты, определяющие прямую наилучшего соответствия.
- В отличие от линейной функции $p(x)$, которая увеличивается или уменьшается на постоянную величину при изменении x на постоянное значение, экспоненциальная функция уменьшается или увеличивается на постоянное отношение при изменении x на постоянное значение.
- Чтобы подогнать экспоненциальное уравнение под данные, можно придерживаться той же процедуры, что и для линейного уравнения, — найти пару (q, r), дающую экспоненциальную функцию qe^{rx}, которая минимизирует функцию потерь.

15 Классификация данных и логистическая регрессия

В этой главе

- ✓ Задача классификации и оценка классификаторов.
- ✓ Поиск границ решения для классификации двух типов данных.
- ✓ Аппроксимация классифицированных наборов данных с помощью логистических функций.
- ✓ Разработка функции потерь для логистической регрессии.
- ✓ Применение градиентного спуска для поиска логистической функции лучшего соответствия.

Классификация — это один из наиболее важных классов задач машинного обучения, и мы сосредоточимся на ней в последних двух главах этой книги. Суть задачи классификации заключается в том, чтобы по одному или нескольким фрагментам исходных данных определить, какой объект представляет каждый из них. Например, представьте алгоритм, просматривающий все сообщения электронной почты, поступающие в наш почтовый ящик, и классифицирующий каждое из них как содержательное сообщение или нежелательный спам. Еще более впечатляющим примером мог бы служить алгоритм классификации для анализа результатов медицинских обследований, определяющий наличие или отсутствие доброкачественных или злокачественных опухолей.

Мы можем создавать алгоритмы машинного обучения для классификации, которые чем больше реальных данных получают, тем детальнее изучают предмет и тем лучше справляются с задачей классификации. Например, каждый раз, когда пользователь электронной почты помечает электронное письмо как спам или рентгенолог обнаруживает злокачественную опухоль, эти данные могут быть переданы алгоритму для улучшения его калибровки.

В этой главе мы продолжим применять тот же простой набор данных, что и в предыдущей главе, с информацией о пробеге и цене подержанных автомобилей. Только теперь данные будут содержать информацию не об одной модели автомобиля, как в предыдущей главе, а о двух: Toyota Prius и седанах BMW 5-й серии. Мы обучим наш алгоритм различать эти модели, основываясь только на числовых данных о пробеге и цене, а также на наборе меток. В отличие от регрессионной модели, которая принимает число и выдает другое число, классификационная модель принимает вектор и выдает число в диапазоне от 0 до 1, отражающее степень уверенности в том, что вектор представляет BMW, а не Prius (рис. 15.1).

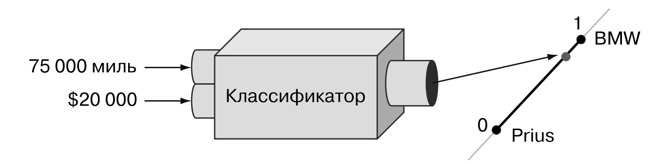


Рис. 15.1. Классификатор получает вектор из двух чисел — пробега и цены подержанного автомобиля — и возвращает число, отражающее его уверенность в том, что этот вектор соответствует автомобилю BMW

Несмотря на то что классификация получает и выдает иные данные, чем регрессия, мы можем построить классификатор регрессионного типа. Алгоритм, реализуемый в этой главе, называется *логистической регрессией*. Для обучения алгоритма используем известный набор данных с пробегом и ценой подержанных автомобилей, включающий метку 1, если данные относятся к автомобилю BMW, и 0, если к Prius. В табл. 15.1 показаны некоторые примеры данных из этого набора.

Таблица 15.1. Примеры данных, используемых для обучения алгоритма

Пробег, миль	Цена, долларов	Это BMW?
110 890	13 995	1
94 133	13 982	1
70 000	9 900	0
46 778	14 599	1
84 507	14 998	0
…	…	…

Нам нужна функция, принимающая значения из первых двух столбцов и возвращающая результат — число между нулем и единицей, близкое к правильной модели автомобиля. Я познакомлю вас с особым типом функций, называемых *логистическими*. Наша функция будет принимать два числа и возвращать одно число, которое всегда находится между 0 и 1. Классификационная функция — это логистическая функция наилучшего соответствия предоставленным выборочным данным.

Наша классификационная функция не всегда будет давать правильный ответ, впрочем, как и человек. Седаны BMW 5-й серии — это роскошные автомобили, поэтому предполагается, что цена Prius будет ниже цены BMW с таким же пробегом. Вопреки нашим ожиданиям, последние две строки в табл. 15.1 показывают одинаковую цену Prius и BMW, при этом Prius имеет почти вдвое больший пробег, чем BMW. Из-за случайных образцов данных, подобных этим, мы не ждем, что логистическая функция выдаст ровно 1 или 0 для каждого BMW или Prius. Иногда функция может вернуть 0,51, сообщая тем самым, что она не уверена, но данные с большей вероятностью представляют BMW.

В предыдущей главе мы видели, что выбранная нами линейная функция определяется двумя параметрами, a и b, в формуле $f(x) = ax + b$. Логистические функции, которые будут применяться в этой главе, имеют три параметра, поэтому задача логистической регрессии сводится к поиску трех чисел, максимально приближающих логистическую функцию к предоставленным выборочным данным. Мы создадим специальную функцию потерь для логистической функции и, используя градиентный спуск, найдем три параметра, минимизирующих функцию потерь. Нам предстоит сделать много шагов, но, к счастью, все они подобны тем, что были сделаны в предыдущей главе, так что это будет полезное повторение материала для тех, кто впервые знакомится с регрессией.

Основная часть главы будет посвящена написанию алгоритма логистической регрессии, но перед этим мы потратим немного времени на знакомство с процессом классификации. Прежде чем обучать компьютер выполнять классификацию, нужно найти способ, позволяющий количественно оценить качество классификации. Затем, построив модель логистической регрессии, мы сможем оценить, насколько хорошо она работает.

15.1. ОЦЕНКА ФУНКЦИИ КЛАССИФИКАЦИИ НА РЕАЛЬНЫХ ДАННЫХ

Посмотрим, насколько уверенно мы можем идентифицировать автомобили BMW в нашем наборе данных, используя простой критерий. А именно, если цена подержанного автомобиля превышает 25 000 долларов, то это, вероятно, слишком дорого для Prius (в конце концов, за эти деньги можно приобрести совершенно новый Prius). Если цена выше 25 000 долларов, то мы скажем, что это BMW,

в противном случае — что это Prius. Такой классификатор легко реализовать в виде функции на языке Python:

```
def bmw_finder(mileage,price):
    if price > 25000:
        return 1
    else:
        return 0
```

Качество этого классификатора, возможно, не особенно высокое, потому что вполне может случиться, что BMW с большим пробегом будут продаваться по цене меньше 25 000 долларов. Но не будем строить догадки — можно просто определить, насколько хорошо этот классификатор работает на реальных данных.

В этом разделе измерим качество нашего алгоритма, написав функцию `test_classifier`, которая принимает функцию классификации, например `bmw_finder`, а также набор данных для тестирования. Набор данных будет представлен массивом кортежей с пробегами, ценами и метками 1 или 0, определяющими модель автомобиля — BMW или Prius. Функция `test_classifier` будет возвращать процентное значение, сообщающее, сколько автомобилей было идентифицировано правильно. В конце главы, реализовав логистическую регрессию, мы сможем передать функцию логистической классификации в `test_classifier` и получить оценку качества ее работы.

15.1.1. Загрузка данных об автомобилях

Написать функцию `test_classifier` будет проще, если мы сначала загрузим данные об автомобилях. Чтобы не возиться с загрузкой данных с сайта CarGraph.com или из файла, я решил упростить вам задачу, поместив данные в файл на Python `car_data.py`, который вы найдете в примерах с исходным кодом для книги. В нем определяются два массива данных: один для Prius и один для BMW. Эти два массива можно импортировать так:

```
from car_data import bmws, priuses
```

Заглянув в файл, вы увидите исходные данные об автомобилях BMW и Prius, и этих данных больше, чем нам нужно. Сейчас нас интересуют только пробег и цена каждой машины, и мы знаем модель всех машин, поскольку они находятся в списке. Например, список автомобилей BMW начинается так:

```
[('bmw', '5', 2013.0, 93404.0, 13999.0, 22.09145859494213),
 ('bmw', '5', 2013.0, 110890.0, 13995.0, 22.216458611342592),
 ('bmw', '5', 2013.0, 94133.0, 13982.0, 22.09145862741898),
 ...
```

Каждый кортеж представляет один автомобиль, выставленный на продажу, а пробег и цена задаются четвертым и пятым элементами кортежа соответственно. В `car_data.py` они преобразуются в объекты `Car`, поэтому можно, например,

написать `car.price` вместо `car[4]`. Мы можем сформировать список `all_car_data` определенной формы, выбрав нужные элементы из кортежей BMW и Prius:

```
all_car_data = []
for bmw in bmws:
    all_car_data.append((bmw.mileage,bmw.price,1))
for prius in priuses:
    all_car_data.append((prius.mileage,prius.price,0))
```

После выполнения этого кода в `all_car_data` будет храниться список, в начале которого находятся данные об автомобилях BMW, а в конце — о Prius с метками 1 и 0 соответственно:

```
>>> all_car_data
[(93404.0, 13999.0, 1),
 (110890.0, 13995.0, 1),
 (94133.0, 13982.0, 1),
 (46778.0, 14599.0, 1),
 ...
 (45000.0, 16900.0, 0),
 (38000.0, 13500.0, 0),
 (71000.0, 12500.0, 0)]
```

15.1.2. Оценка функции классификации

Получив данные в подходящем формате, можно приступать к функции `test_classifier`. Задача `bmw_finder` — оценить пробег и цену автомобиля и сообщить, относится ли он к модели BMW. Если ответ положительный, функция возвращает 1, иначе — 0. Вполне вероятно, что для каких-то автомобилей `bmw_finder` даст неверный ответ. Если она предсказывает, что автомобиль — это BMW (возвращая 1), а на самом деле это Prius, то мы будем считать такой результат *ложноположительным*. Если он предсказывает, что автомобиль — это Prius (возвращая 0), а на самом деле это BMW, то мы будем считать такой результат *ложноотрицательным*. Если функция правильно идентифицирует BMW или Prius, мы будем называть результат *истинно положительным* или *истинно отрицательным* соответственно.

Чтобы проверить качество функции классификации на наборе данных `all_car_data`, передадим ей каждый кортеж с пробегом и ценой из этого списка и посмотрим, соответствует ли результат 1 или 0 фактической метке. Вот как это выглядит в коде:

```
def test_classifier(classifier, data):
    trues = 0
    falses = 0
    for mileage, price, is_bmw in data:
        if classifier(mileage, price) == is_bmw:
            trues += 1
        else:
            falses += 1
    return trues / (trues + falses)
```

Прибавить 1 к счетчику trues, если автомобиль классифицирован верно

Иначе прибавить 1 к счетчику falses

Применив эту функцию к функции классификации `bmw_finder` и набору данных `all_car_data`, вы увидите, что она имеет точность классификации 59 %:

```
>>> test_classifier(bmw_finder, all_car_data)
0.59
```

Это не так уж плохо: большинство автомобилей классифицировано верно. Но, как вы увидите далее, можно добиться большего! В следующем разделе мы построим график набора данных, чтобы понять, в чем основная проблема функции `bmw_finder`. График поможет увидеть, как можно улучшить функции логистической классификации и добиться от нее более высокой точности.

15.1.3. Упражнения

Упражнение 15.1. Добавьте в функцию `test_classifier` вывод количества истинно положительных, истинно отрицательных, ложноположительных и ложноотрицательных результатов. Еще раз оцените классификатор `bmw_finder`. Что вы можете сказать о его качестве?

Решение. Вместо простого суммирования верных и неверных прогнозов можем отдельно суммировать истинные и ложные положительные и отрицательные результаты:

```
def test_classifier(classifier, data, verbose=False):
    true_positives = 0
    true_negatives = 0
    false_positives = 0
    false_negatives = 0

    for mileage, price, is_bmw in data:
        predicted = classifier(mileage,price)
        if predicted and is_bmw:
            true_positives += 1
        elif predicted:
            false_positives += 1
        elif is_bmw:
            false_negatives += 1
        else:
            true_negatives += 1

    if verbose:
        print("true positives %f" % true_positives)
        print("true negatives %f" % true_negatives)
        print("false positives %f" % false_positives)
        print("false negatives %f" % false_negatives)

    total = true_positives + true_negatives

    return total / len(data)
```

Определяет необходимость вывода результатов (иногда это может быть нежелательно)

Теперь будут накапливаться четыре суммы

В зависимости от принадлежности автомобиля к той или иной модели и правильности классификации увеличивается один из четырех счетчиков

Вывод значений счетчиков

Вернуть количество правильных классификаций (истинно положительных и истинно отрицательных), деленное на количество автомобилей в наборе данных

Для функции bmw_finder будет выведен такой текст:

```
true positives 18.000000
true negatives 100.000000
false positives 0.000000
false negatives 82.000000
```

Поскольку в ходе работы классификатора не отмечено ни одного ложноположительного результата, можно утверждать, что он всегда правильно определяет, когда автомобиль *не* является BMW. Но мы пока не можем гордиться своей функцией, потому что она утверждает, что большинство автомобилей не являются автомобилями BMW, хотя это не так! В следующем упражнении вы сможете ослабить ограничение и повысить общую вероятность успеха.

Упражнение 15.2. Попробуйте усовершенствовать функцию bmw_finder, чтобы повысить точность ее прогнозов, и используйте функцию test_classifier, чтобы убедиться, что усовершенствованная функция показывает точность выше 59 %.

Решение. Если вы выполнили предыдущее упражнение, то не могли не заметить, что bmw_finder слишком часто заявляет, что рассматриваемый автомобиль — это не BMW. Можно попробовать снизить ценовой порог до 20 000 долларов и посмотреть, даст ли это положительный эффект:

```
def bmw_finder2(mileage,price):
    if price > 20000:
        return 1
    else:
        return 0
```

И действительно, снижение порога повысило показатель успеха bmw_finder до 73,5 %:

```
>>> test_classifier(bmw_finder2, all_car_data)
0.735
```

15.2. ИЗОБРАЖЕНИЕ ГРАНИЦ РЕШЕНИЯ

Прежде чем реализовать функцию логистической регрессии, рассмотрим еще один способ оценки качества классификации. Поскольку подержанные автомобили определяются двумя числовыми параметрами — пробегом и ценой, мы можем рассматривать их как двухмерные векторы и изобразить точками на двухмерной плоскости. Построив такой график, сможем получить более

полное представление о том, где функция классификации проводит черту между моделями BMW и Prius, и понять, как ее улучшить. Оказывается, применение функции `bmw_finder` сродни рисованию прямой линии на двухмерной плоскости, при этом любая точка над линией называется BMW, а любая точка под ней — Prius.

В этом разделе мы используем библиотеку Matplotlib, с ее помощью нарисуем график и посмотрим, где `bmw_finder` проводит разделительную линию между BMW и Prius. Эта линия называется *границей решения*, потому что нахождение точки по ту или иную сторону от прямой помогает решить, к какому классу ее отнести. Посмотрев на график с данными об автомобилях, мы сможем понять, где лучше провести разделительную линию, а это в свою очередь поможет определить улучшенную версию функции `bmw_finder` и оценить, насколько лучше она справляется с классификацией.

15.2.1. Изображение пространства автомобилей

Все автомобили в нашем наборе данных характеризуются значениями пробега и цены, но некоторые из них представляют BMW, а некоторые — Prius в зависимости от значения метки, 1 или 0. Чтобы график получился нагляднее, сделаем автомобили BMW и Prius визуально различимыми.

Вспомогательная функция `plot_data` (вы найдете ее в примерах исходного кода) принимает весь список с данными об автомобилях и автоматически отображает BMW крестиками, а Prius — кружками:

```
>>> plot_data(all_car_data)
```

Получившийся график показан на рис. 15.2.

На графике видно, что в общем случае автомобили BMW стоят дороже, чем Prius: большинство крестиков, представляющих BMW, находятся выше по оси цены. Это оправдывает нашу стратегию классификации более дорогих автомобилей как BMW. В частности, мы провели линию по цене 25 000 долларов (рис. 15.3). На графике она отделяет верхнюю часть с более дорогими автомобилями от нижней с менее дорогими.

Это граница решения. Каждый крестик над прямой линией правильно идентифицируется как BMW, а каждый кружок под ней — правильно классифицируется как Prius. Все остальные точки классифицированы неправильно. Очевидно, что, сдвинув границу решения, можно повысить точность. Давайте попробуем.

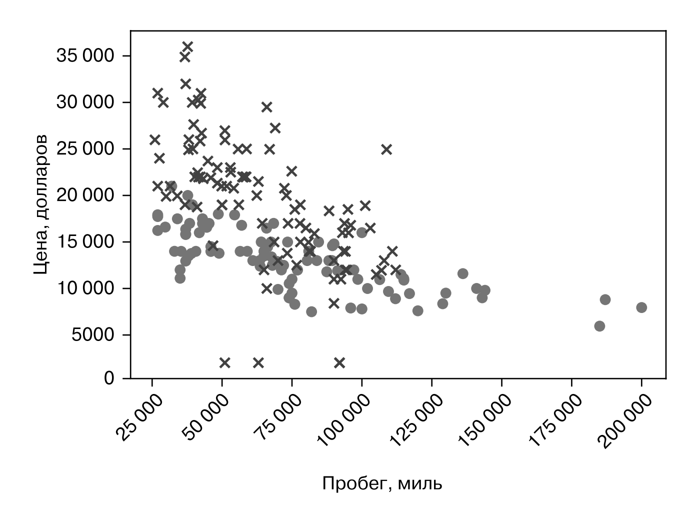


Рис. 15.2. График зависимости цены от пробега для всех автомобилей в наборе данных, где каждый автомобиль BMW обозначен крестиком, а каждый Prius — кружком

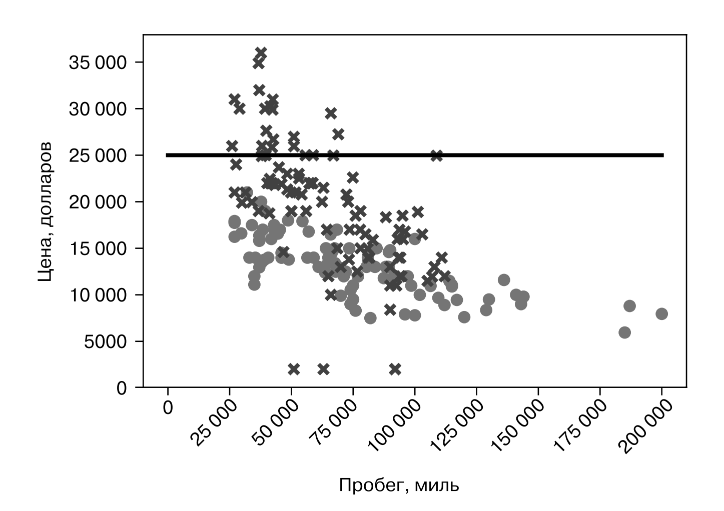


Рис. 15.3. Здесь показана линия принятия решения на фоне данных об автомобилях

15.2.2. Определение лучшей границы решения

Основываясь на графике на рис. 15.3, мы могли бы опустить линию и правильно классифицировать еще несколько автомобилей BMW, не увеличив количество неверно классифицированных автомобилей Prius. На рис. 15.4

показано, где будет располагаться граница решения, если пороговая цена снизится до 21 000 долларов.

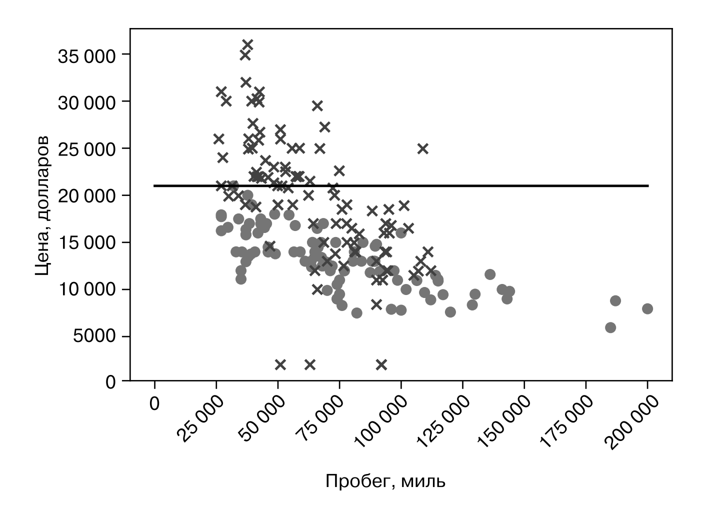


Рис. 15.4. Опускание линии границы решения увеличивает точность

Порог в 21 000 долларов может быть хорошей границей для автомобилей с небольшим пробегом, но чем больше пробег, тем ниже порог. Например, похоже, что большинство BMW с пробегом 75 000 миль или более стоят меньше 21 000 долларов. Чтобы учесть это, можно сделать пороговую цену *зависимой от пробега*. Геометрически это означает проведение линии с наклоном вниз (рис. 15.5).

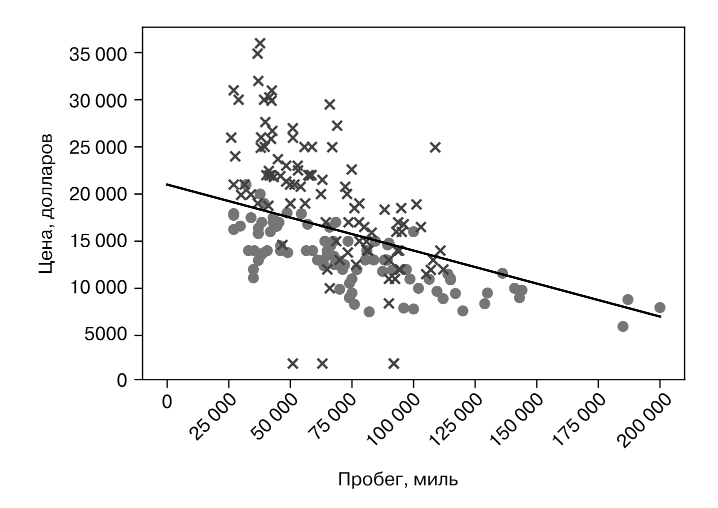


Рис. 15.5. Использование границы решения с наклоном вниз

Эта прямая задается функцией $p(x) = 21\,000 - 0{,}07x$, где p — цена, а x — пробег. В этом уравнении нет ничего особенного — я просто поиграл с числами, пока не получил более или менее подходящую прямую. Но похоже, что она правильно классифицирует еще больше автомобилей BMW, чем раньше, хотя и за счет увеличения числа ложноположительных результатов, когда автомобили Prius неправильно классифицируются как BMW. Вместо того чтобы просто рассматривать эти границы решений, можно превратить их в функции классификации и оценить их эффективность.

15.2.3. Реализация функции классификации

Чтобы превратить границу решения в функцию классификации, нужно написать функцию на Python, которая принимает данные о пробеге и цене автомобиля и возвращает единицу или ноль в зависимости от того, где находится точка — выше или ниже прямой. Это означает, что нужно взять заданный пробег, включить его в функцию границы решения $p(x)$, чтобы определить пороговую цену, и сравнить результат с заданной ценой. Вот как это выглядит:

```
def decision_boundary_classify(mileage,price):
    if price > 21000 - 0.07 * mileage:
        return 1
    else:
        return 0
```

Протестировав ее, мы видим, что она намного точнее первоначального классификатора — 80,5 % автомобилей классифицируются правильно:

```
>>> test_classifier(decision_boundary_classify, all_car_data)
0.805
```

Неплохо!

Вы можете спросить, почему нельзя просто выполнить градиентный спуск по параметрам, определяющим прямую границы решения. Если 20 000 и 0,07 не дают самой точной границы решения, то, может быть, какая-то пара чисел рядом с ними дадут ее. Это вполне здравая идея. Когда мы реализуем логистическую регрессию, вы увидите, что за кулисами она перемещает границу решения, используя градиентный спуск, пока не найдет лучший вариант.

Есть две важные причины, по которым мы реализуем более сложный алгоритм логистической регрессии, а не градиентный спуск по параметрам a и b функции границы решения $ax + b$. Во-первых, если граница решения близка к вертикали на любом этапе градиентного спуска, то числа a и b могут стать очень большими и вызвать проблемы с округлением. Во-вторых, у нас нет очевидной функции потерь. В следующем разделе увидим, как логистическая регрессия решает обе эти проблемы, позволяя искать наилучшую границу решения с использованием градиентного спуска.

15.2.4. Упражнение

Упражнение 15.3. Мини-проект. Определите границу решения в форме p = const, которая дает наилучшую точность классификации тестовых данных.

Решение. Следующая функция строит функцию-классификатор для любой заданной постоянной пороговой цены. Другими словами, получающийся классификатор возвращает 1, если проверяемый автомобиль стоит больше порогового значения, и 0 — в противном случае:

```
def constant_price_classifier(cutoff_price):
    def c(x,p):
        if p > cutoff_price:
            return 1
        else:
            return 0
    return c
```

Точность этой функции можно оценить, передав ее функции `test_classifier`. Вот вспомогательная функция, автоматизирующая эту проверку для любой заданной пороговой цены:

```
def cutoff_accuracy(cutoff_price):
    c = constant_price_classifier(cutoff_price)
    return test_classifier(c,all_car_data)
```

Лучшая пороговая цена находится между двумя ценами в нашем списке. Достаточно проверить каждую из них и посмотреть, является ли она лучшей пороговой ценой. Это можно быстро реализовать на Python, используя функцию `max`. Именованный аргумент `key` позволяет выбрать функцию для максимизации. В данном случае нужно найти в списке цену, которая является наилучшей пороговой ценой, поэтому можно выполнить максимизацию с помощью функции `cutoff_accuracy`:

```
>>> max(all_prices,key=cutoff_accuracy)
17998.0
```

Этот результат говорит, что согласно нашему набору данных 17 998 долларов — лучшая цена, которую можно использовать в качестве пороговой, принимая решение о том, к какой модели отнести автомобиль — BMW 5-й серии или Prius. Этот классификатор оказался довольно точным для нашего набора данных, показав точность 79,5 %:

```
>>> test_classifier(constant_price_classifier(17998.0), all_car_data)
0.795
```

15.3. КЛАССИФИКАЦИЯ КАК ЗАДАЧА РЕГРЕССИИ

Мы можем представить классификацию как задачу регрессии. Для этого нужно лишь создать функцию, которая принимает пробег и цену автомобиля и возвращает число, оценивающее вероятность того, что этот автомобиль — BMW, а не Prius. В этом разделе мы реализуем функцию `logistic_classifier`, внешне очень похожую на классификаторы, созданные до сих пор. Она будет принимать пробег и цену и возвращать число, позволяющее классифицировать автомобиль как BMW или Prius. Единственное отличие в том, что возвращаться будет не ноль или единица, а значение между нулем и единицей, определяющее вероятность того, что рассматриваемый автомобиль — BMW.

Это число можно рассматривать как вероятность того, что заданные пробег и цена описывают BMW, или, более абстрактно, степень похожести точки данных на BMW (рис. 15.6).

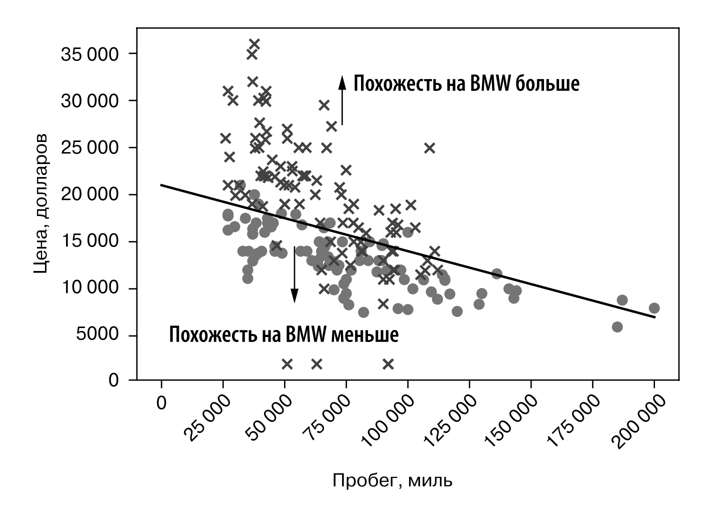


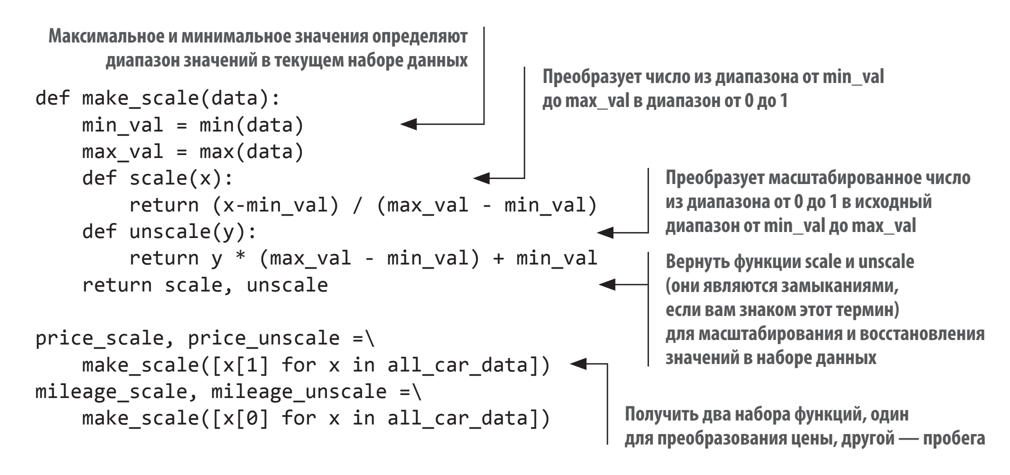
Рис. 15.6. Понятие похожести на BMW описывает, насколько точка на плоскости похожа на BMW

Строительство логистического классификатора начнем с выбора хорошей границы правильного решения. Точки над границей имеют высокую похожесть на BMW и, вероятно, представляют автомобили BMW, поэтому логистическая функция должна возвращать значения, близкие к единице. Точки под границей имеют низкую похожесть на BMW и, скорее всего, представляют автомобили Prius, поэтому функция должна возвращать значения, близкие к нулю. На границе принятия решения значение похожести на BMW будет равно 0,5, что означает: точка данных с одинаковой вероятностью может представлять и BMW, и Prius.

15.3.1. Масштабирование исходных данных об автомобилях

В какой-то момент, решая задачу регрессии, необходимо выполнить рутинную работу, чем мы сейчас и займемся. Как обсуждалось в предыдущей главе, большие значения пробега и цены могут привести к вычислительным ошибкам, поэтому лучше масштабировать их до небольших сопоставимых величин. Лучше всего масштабировать величины пробега и цены так, чтобы они имели значения в диапазоне от нуля до единицы.

Нам нужна возможность масштабировать и восстанавливать масштабированные значения пробега и цены, то есть требуются четыре функции. Чтобы немного упростить задачу, я написал вспомогательную функцию, которая принимает список чисел и возвращает функции для их линейного масштабирования в диапазон от нуля до единицы и восстановления, используя максимальное и минимальное значения в списке. Применение вспомогательной функции ко всему набору данных дает четыре нужных функции:




```
def make_scale(data):
    min_val = min(data)
    max_val = max(data)
    def scale(x):
        return (x-min_val) / (max_val - min_val)
    def unscale(y):
        return y * (max_val - min_val) + min_val
    return scale, unscale

price_scale, price_unscale =\
    make_scale([x[1] for x in all_car_data])
mileage_scale, mileage_unscale =\
    make_scale([x[0] for x in all_car_data])
```

Теперь можем применить эти функции масштабирования к каждой точке данных в списке и получить масштабированную версию набора данных:

```
scaled_car_data = [(mileage_scale(mileage), price_scale(price), is_bmw)
                   for mileage,price,is_bmw in all_car_data]
```

Самое интересное, что внешне график не изменился (рис. 15.7), изменились только значения на осях.

То, что геометрия масштабированного набора данных осталась прежней, должно придать нам уверенности, что хорошую границу решения для этого масштабированного набора данных легко будет преобразовать в хорошую границу решения для исходного набора данных.

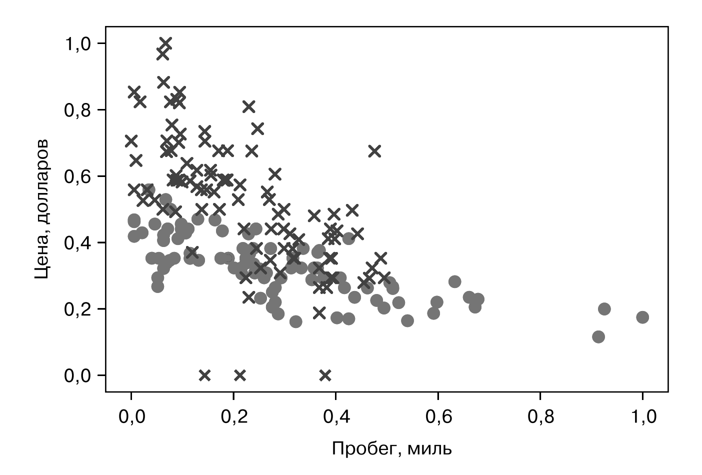


Рис. 15.7. Данные о пробеге и ценах масштабированы так, что все значения находятся в диапазоне от нуля до единицы. Внешне график такой же, как и раньше, но риск ошибок вычисления уменьшился

15.3.2. Оценка похожести автомобиля на BMW

Начнем с границы решения, похожей на ту, что была найдена в предыдущем разделе. Функция $p(x) = 0{,}56 - 0{,}35x$ дает цену на границе решения как функцию пробега. Эта функция довольно близка к той, что я обнаружил в предыдущем разделе, но получена на основе набора масштабированных данных (рис. 15.8).

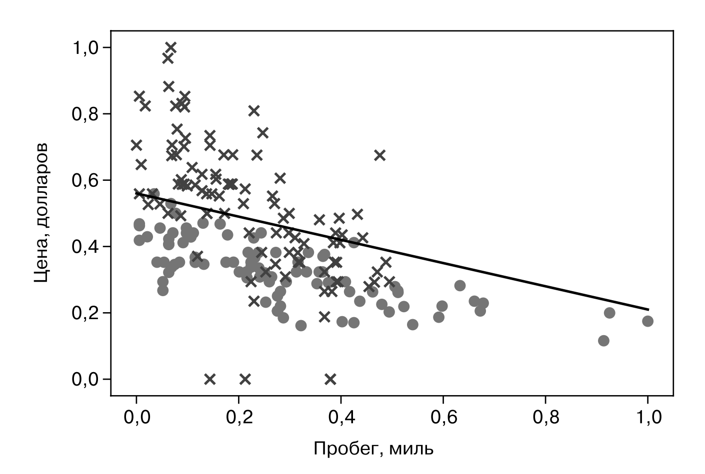


Рис. 15.8. Граница решения $p(x) = 0{,}56 - 0{,}35x$ для набора масштабированных данных

Мы все так же можем тестировать классификаторы, полученные на наборе масштабированных данных, используя функцию `test_classifier`, нужно лишь позаботиться о передаче масштабированных данных вместо оригинала. Эта граница решения дает точность классификации на уровне 78,5 %.

Эту функцию границы решения можно также изменить, чтобы передать степень похожести на BMW точки данных. Для простоты запишем границу решения как

$$p = ax + b,$$

где p — цена, x — пробег, а коэффициенты a и b — наклон и точка пересечения прямой с осью y (в данном случае $a = -0{,}35$ и $b = 0{,}56$) соответственно. Это выражение можно рассматривать не как функцию, а как уравнение, которому удовлетворяют точки (x, p) на границе решения.

Если вычесть $ax + b$ из обеих частей уравнения, то получится еще одно правильное уравнение:

$$p - ax - b = 0.$$

Каждая точка (x, p) на границе решения также удовлетворяет этому уравнению. Другими словами, величина $p - ax - b$ равна нулю для каждой точки на границе решения.

Вот суть этой алгебры: величина $p - ax - b$ является мерой похожести на BMW точки (x, p). Если (x, p) находится выше границы решения, это означает, что p слишком велико для данного значения x, поэтому $p - ax - b > 0$. Напротив, если (x, p) находится ниже границы решения, значит, p слишком мало для данного значения x, поэтому $p - ax - b < 0$. В противном случае выражение $p - ax - b$ точно равно нулю и точка находится прямо на границе выбора между Prius и BMW. При первом прочтении это рассуждение может показаться немного отвлеченным, поэтому в табл. 15.2 перечислены три случая.

Таблица 15.2. Три возможных случая

(x, p) *выше* границы решения	$p - ax - b > 0$	Скорее всего, это BMW
(x, p) *на* границе решения	$p - ax - b = 0$	Это может быть как BMW, так и Prius
(x, p) *ниже* границы решения	$p - ax - b < 0$	Скорее всего, это Prius

Если вы не уверены, что $p - ax - b$ является мерой похожести на BMW, совместимой с границей решения, то убедиться в этом вам поможет тепловая карта $f(x, p) = p - ax - b$ (рис. 15.9). Когда $a = -0{,}35$ и $b = 0{,}56$, функция принимает вид $f(x, p) = p - 0{,}35x - 0{,}56$.

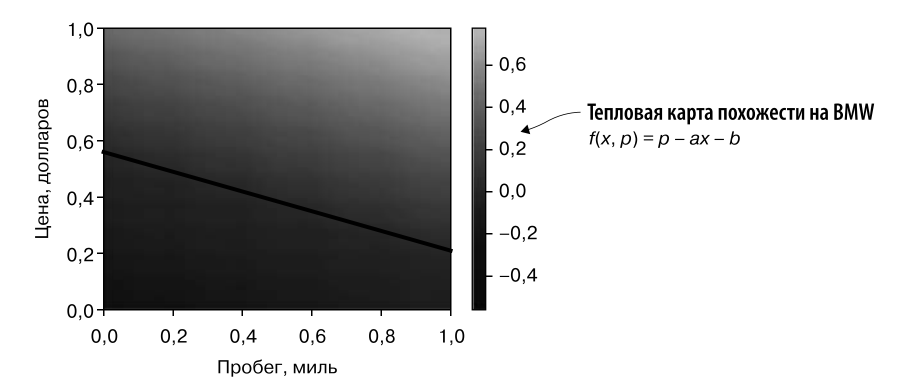


Рис. 15.9. Тепловая карта и границы решения. Выше границы решения находится область более светлых значений (положительных значений похожести на BMW), а ниже — более темных (отрицательных значений похожести на BMW)

Функция $f(x, p)$ *почти* соответствует нашим требованиям. Он принимает пробег и цену и возвращает число, которое тем больше, чем выше вероятность того, что пробег и цена представляют BMW, и тем меньше, чем выше вероятность, что Prius. Единственное, чего не хватает, — ограничения результата диапазоном от нуля до единицы, да и пороговое значение 0, а не 0,5, как хотелось бы. К счастью, есть удобная математическая функция, которую можно использовать для преобразования результата.

15.3.3. Знакомство с сигмоидной функцией

Функция $f(x, p) = p - ax - b$ — линейная, но эта глава не о линейной регрессии! Ее главная тема — *логистическая регрессия*, поэтому требуется логистическая функция. Далее приводится самая простая логистическая функция, которую часто называют *сигмоидной*:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Эту функцию можно реализовать на Python с помощью функции `exp`, которая заменяет e^x, где e = 2,71828..., и является константой, использованной в роли основания экспоненциальной функции ранее:

```
from math import exp
def sigmoid(x):
    return 1 / (1+exp(-x))
```

Ее график показан на рис. 15.10.

Функция обозначается греческой буквой σ («сигма»), потому что σ — это греческая версия буквы S, а график $\sigma(x)$ немного похож на букву S. Иногда термины

«логистическая функция» и *«сигмоидная функция»* используются как взаимозаменяемые и означают функцию, подобную показанной на рис. 15.10, которая плавно возрастает с переменным ускорением. В этой и следующей главе, говоря о сигмоидной функции, я буду подразумевать конкретную функцию $\sigma(x)$.

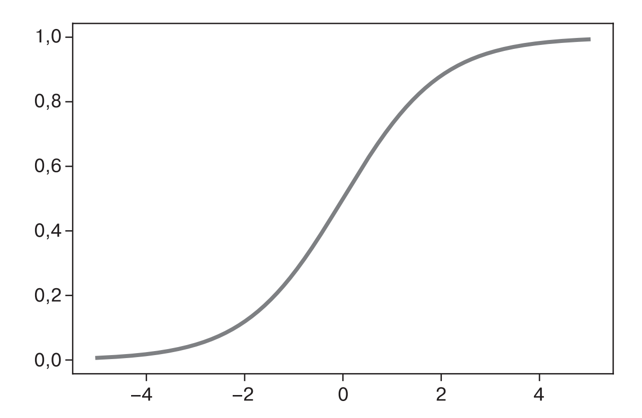


Рис. 15.10. График сигмоидной функции $\sigma(x)$

Вам не нужно беспокоиться о том, как определяется эта функция, важно лишь понимать форму графика и его значение. Эта функция преобразует любое входное число в значение от нуля до единицы, при этом большие отрицательные числа дают результаты ближе к нулю, а большие положительные числа — ближе к единице. Результат $\sigma(0)$ равен 0,5. Функцию σ можно рассматривать как преобразование диапазона от $-\infty$ до ∞ в более удобный диапазон от нуля до единицы.

15.3.4. Комбинирование сигмоидной функции с другими функциями

А теперь вернемся к функции $f(x, p) = p - ax - b$. Как мы видели, она принимает значения пробега и цены и возвращает число, определяющее степень похожести на BMW. Это число может быть большим, положительным или отрицательным, а значение, равное нулю, указывает на то, что рассматриваемая точка данных находится точно на границе между BMW и Prius.

Нам нужно, чтобы функция возвращала значение от нуля до единицы (желательно как можно ближе к ним), представляющее автомобиль, который с большей вероятностью можно отнести к Prius или BMW соответственно, и значение 0,5, представляющее автомобиль, который с равной вероятностью можно отнести и к Prius, и к BMW. Для этого достаточно преобразовать результат $f(x, p)$ так, чтобы он находился в ожидаемом диапазоне, то есть обработать его сигмоидной функцией $\sigma(x)$, как показано на рис. 15.11. То есть нам нужна функция $\sigma(f(x, p))$, где x и p — пробег и цена.

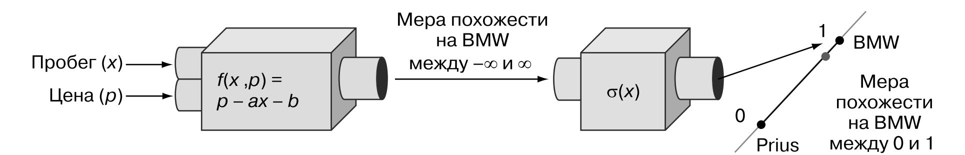
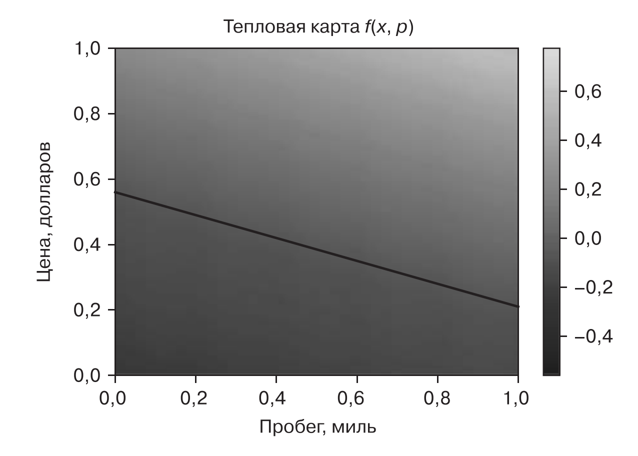


Рис. 15.11. Схематическое изображение объединения функции похожести на BMW $f(x, p)$ с сигмоидной функцией $\sigma(x)$

Обозначим полученную функцию $L(x, p)$, то есть $L(x, p) = \sigma(f(x, p))$. Реализовав функцию $L(x, p)$ на Python и построив ее тепловую карту (рис. 15.12), можно заметить, что она увеличивается в том же направлении, что и $f(x, p)$, но дает другие значения.



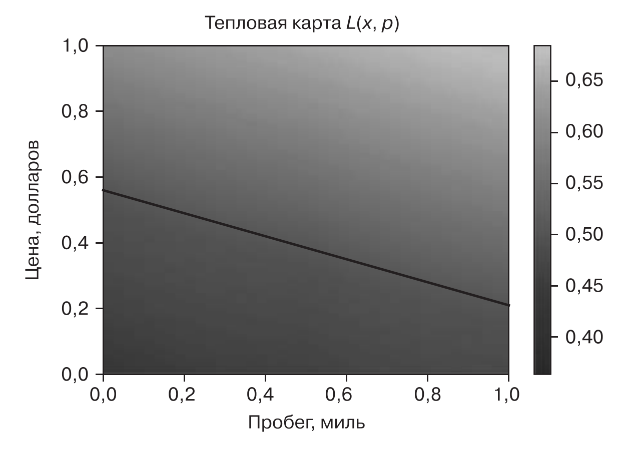


Рис. 15.12. Тепловые карты выглядят практически одинаковыми, но значения функций получаются немного разными

При взгляде на эту диаграмму возникает вопрос: к чему такие сложности с передачей функции похожести на BMW в сигмоидную функцию? С этой точки зрения обе функции выглядят почти одинаково. Однако, если построить их графики в виде двухмерных поверхностей в трехмерном пространстве (рис. 15.13), то можно увидеть, что сигмоида имеет изогнутую форму.

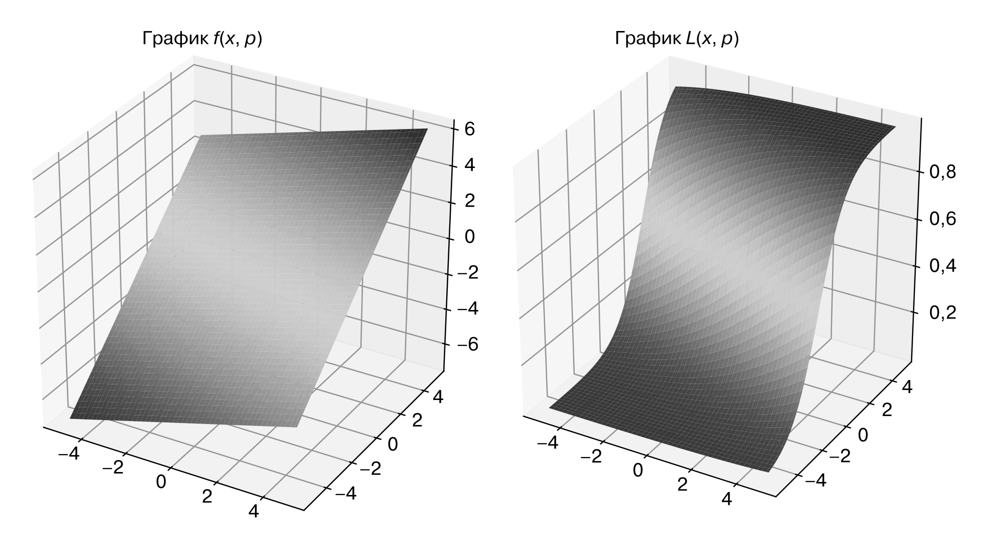
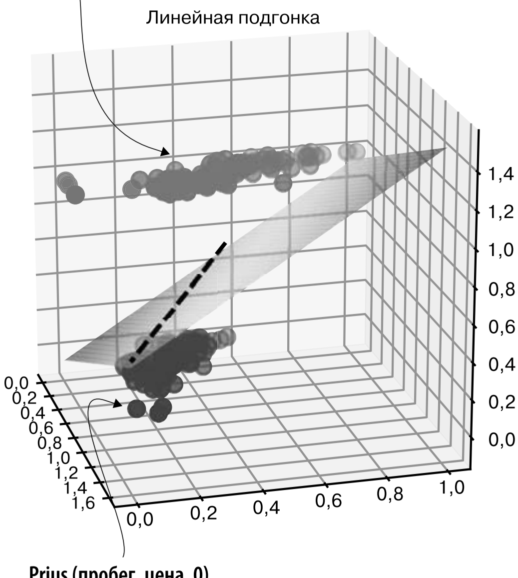


Рис. 15.13. Функция $f(x, p)$ описывает плоскую поверхность с наклоном вверх, а $L(x, p)$ изгибается при движении вверх от минимального значения 0 до максимального значения 1

Справедливости ради отмечу, что мне пришлось немного уменьшить масштаб в пространстве (x, p), чтобы кривизна стала видна отчетливее. Дело в том, что если модель автомобиля обозначить как 0 или 1, то значения функции $L(x, p)$ действительно стремятся к этим числам, тогда как значения $f(x, p)$ уходят в бесконечность в обе стороны!

На рис. 15.14 показаны две утрированные диаграммы, чтобы помочь вам понять, что я имею в виду. Помните, что в своем наборе данных `scaled_car_data` мы представили Prius тройками формы (пробег, цена, 0), а BMW — тройками формы (пробег, цена, 1). Мы можем интерпретировать их как точки в трехмерном пространстве, где автомобили BMW находятся на плоскости $z = 1$, а автомобили Prius — на плоскости $z = 0$. Отобразив `scaled_car_data` на трехмерной диаграмме, можно увидеть, что линейная функция не может приблизиться ко многим точкам данных так, как это может логистическая функция.



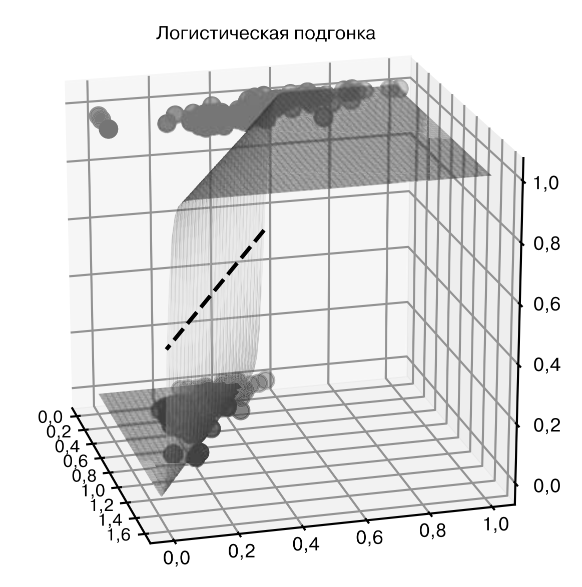


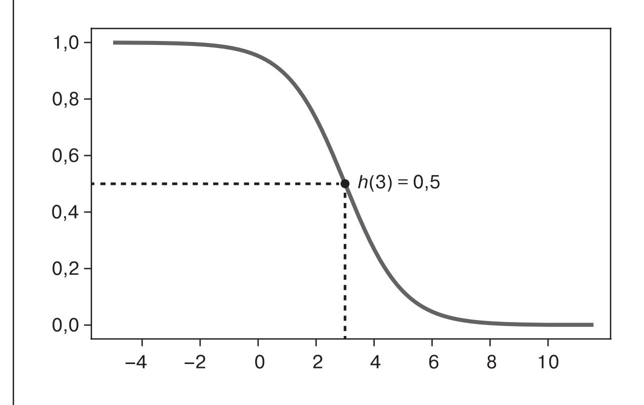
Рис. 15.14. График линейной функции в трехмерном пространстве не может приблизиться к точкам данных, как это может график логистической функции

Используя функции в форме $L(x, p)$, мы действительно можем надеяться *подогнать* их под данные. А как это сделать, посмотрим в следующем разделе.

15.3.5. Упражнения

Упражнение 15.4. Найдите функцию $h(x)$, которая для больших положительных x возвращает значение, близкое к 0, для больших отрицательных x — значение, близкое к 1, а для $x = 3$ — значение 0,5.

Решение. Функция $y(x) = 3 - x$ имеет $y(3) = 0$ и стремится к положительной бесконечности, когда x имеет большое отрицательное значение, и к отрицательной бесконечности, когда x имеет большое положительное значение. Это означает, что если передать результат $y(x)$ сигмоидной функции, то мы получим функцию с желаемыми свойствами, в частности, $h(x) = \sigma(y(x)) = \sigma(3 - x)$. График этой функции показан далее, чтобы вы могли убедиться сами.



Упражнение 15.5. Мини-проект. На самом деле $f(x, p)$ имеет нижнюю границу, потому что x и p не могут быть отрицательными (отрицательные пробег и цена не имеют смысла). Можете ли вы вычислить наименьшее значение f, которое может быть получено для автомобиля?

Решение. Согласно тепловой карте функция $f(x, p)$ уменьшается по мере движения вниз и влево. Уравнение также подтверждает это: если уменьшить x или p, то значение $f = p - ax - b = p + 0{,}35x - 0{,}56$ уменьшится. Следовательно, функция $f(x, p)$ достигнет минимального значения при $(x, p) = (0, 0)$, а это $f(0, 0) = -0{,}056$.

15.4. ИССЛЕДОВАНИЕ ПРОСТРАНСТВА ВОЗМОЖНЫХ ЛОГИСТИЧЕСКИХ ФУНКЦИЙ

Быстро обобщим сделанные нами шаги. Нанеся на график точки из своего набора данных, обозначающие пробег и цену автомобилей Prius и BMW, мы попытались провести линию, разделяющую эти модели, которая называется границей решения и определяет правило, помогающее отличить Prius от BMW. Мы записали границу решения в виде линейной функции $p(x) = ax + b$, и оказалось, что наиболее оптимальными значениями для a и b являются числа –0,35 и 0,56. Они позволяют классифицировать данные с точностью около 80 %.

Преобразовав эту функцию в форму $f(x, p) = p - ax - b$, мы выяснили, что функция, принимающая величину пробега и цену (x, p), возвращает число больше нуля, если точка данных классифицируется как принадлежащая к категории BMW, и меньше нуля, если как принадлежащая к категории Prius. На границе решения $f(x, p)$ возвращает ноль, а это означает, что точка данных с равной вероятностью может представлять BMW или Prius. Поскольку мы обозначили автомобили BMW меткой 1, а Prius — меткой 0, нам понадобилась версия $f(x, p)$, которая возвращала бы значения от нуля до единицы, где значение 0,5 представляло бы равную вероятность выбора между BMW и Prius. Преобразовав результат f в сигмоидную функцию, мы получили новую функцию $L(x, p) = \sigma(f(x, p))$, удовлетворяющую этому требованию.

Но нам нужна не функция $L(x, p)$, которую я получил, подобрав наилучшую границу решения, а функция $L(x, p)$, которая *лучше всего соответствует данным*. На пути к этой цели мы увидим, что есть три параметра, с помощью которых можно управлять поведением логистической функции, которая принимает двухмерные векторы и возвращает числа от нуля до единицы, а также имеет границу решения $L(x, p) = 0{,}5$ в виде прямой линии. Мы напишем на Python функцию `make_logistic(a,b,c)`, которая принимает три параметра, a, b и c, и возвращает определяемую ими логистическую функцию. Так же, как мы исследовали двухмерное пространство пар (a, b) для выбора линейной функции в главе 14, исследуем трехмерное пространство значений (a, b, c) для определения логистической функции (рис. 15.15).

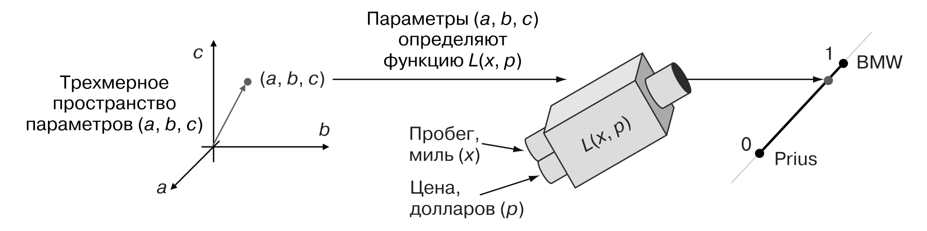


Рис. 15.15. Исследование трехмерного пространства значений параметров (*a*, *b*, *c*) для определения функции *L*(*x*, *p*)

Затем создадим функцию потерь, похожую на ту, что мы создали для линейной регрессии. Функция потерь, которую назовем `logistic_cost(a,b,c)`, будет принимать параметры a, b и c, определяющие логистическую функцию, и возвращать одно число, оценивающее соответствие логистической функции набору данных об автомобилях. Функцию `logistic_cost` необходимо реализовать так, чтобы меньшие возвращаемые значения соответствовали лучшим прогнозам логистической функции.

15.4.1. Параметризация логистических функций

Первая задача — определить общий вид логистической функции $L(x, p)$, значения которой находятся в диапазоне от нуля до единицы, а граница решения $L(x, p) = 0{,}5$ — прямая линия. Мы вплотную подошли к этому в предыдущем разделе, начав с границы решения $p(x) = ax + b$ и реконструировав на ее основе логистическую функцию. Единственная проблема заключается в том, что линейная функция в виде $ax + b$ неспособна представлять произвольную прямую на плоскости. Например, на рис. 15.16 показан набор данных, для которого разумной выглядит вертикальная граница решения $x = 0{,}6$. Однако такую прямую нельзя представить в виде $p = ax + b$.

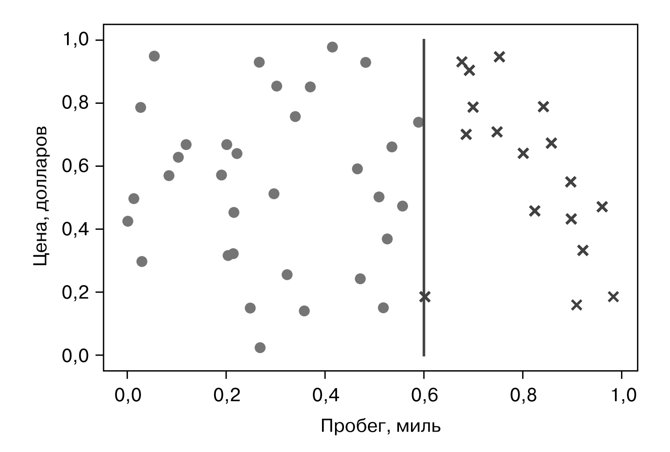


Рис. 15.16. Вертикальная граница решения может иметь смысл, но ее нельзя представить в виде $p = ax + b$

Для общего случая подходит уравнение прямой, с которым мы познакомились в главе 7, — $ax + by = c$. Поскольку наши переменные называются x и p, запишем уравнение так: $ax + bp = c$. Для такого уравнения функция $z(x, p) = ax + bp - c$ дает ноль для точек на прямой, положительные значения — для точек с одной стороны и отрицательные — для точек с другой стороны. Для нас та сторона, где $z(x, p)$ дает положительные значения, — это сторона BMW, а где $z(x, p)$ дает отрицательные значения, — сторона Prius.

Передав $z(x, p)$ в сигмовидную функцию, получим общую логистическую функцию $L(x, p) = \sigma(z(x, p))$, дающую $L(x, p) = 0{,}5$ для точек на прямой, где $z(x, p) = 0$. Другими словами, функция $L(x, p) = \sigma(ax + bp - c)$ — это искомая общая форма. Ее легко реализовать в коде на Python и получить функцию с тремя аргументами, a, b и c, которая возвращает соответствующую логистическую функцию $L(x, p) = \sigma(ax + bp - c)$:

```
def make_logistic(a,b,c):
    def l(x,p):
        return sigmoid(a*x + b*p - c)
    return l
```

Следующий шаг — определить, насколько эта функция соответствует нашему набору данных `scaled_car_data`.

15.4.2. Оценка качества соответствия логистической функции

Для любого автомобиля BMW в списке `scaled_car_data` имеется запись в форме $(x, p, 1)$, а для каждого Prius — запись в форме $(x, p, 0)$, где x и p обозначают масштабированные пробег и цену соответственно. Если применить логистическую функцию $L(x, p)$ к значениям x и p, получим результат между нулем и единицей.

Чтобы измерить ошибку (или потерю) функции L, нужно найти, насколько ее значение отличается от правильного, равного нулю или единице. Сложив все эти ошибки, вы получите общее значение, показывающее, насколько далеки прогнозы функции $L(x, p)$ от фактических данных. Вот как это выглядит на Python:

```
def simple_logistic_cost(a,b,c):
    l = make_logistic(a,b,c)
    errors = [abs(is_bmw-l(x,p))
              for x,p,is_bmw in scaled_car_data]
    return sum(errors)
```

Эта функция хорошо справляется с вычислением ошибки, но этого недостаточно, чтобы обеспечить сходимость градиентного спуска к наилучшим значениям a, b и c. Я не буду вдаваться в исчерпывающее объяснение, почему это так, но постараюсь дать общее представление.

Предположим, у нас есть две логистические функции, $L_1(x, p)$ и $L_2(x, p)$, и мы хотим сравнить качество прогнозирования обеих. Допустим, обе они получают одну и ту же точку данных $(x, p, 0)$, представляющую автомобиль Prius. Предположим также, что $L_1(x, p)$ возвращает 0,99, что больше 0,5, поэтому ошибочно предсказывает, что это автомобиль BMW. Ошибка для этой точки составляет $|0 - 0{,}99| = 0{,}99$. Если логистическая функция $L_2(x, p)$ дает значение 0,999, то она с большей уверенностью предсказывает, что это автомобиль BMW, и еще более ошибочна. При этом ошибка будет составлять всего $|0 - 0{,}999| = 0{,}999$, не сильно отличаясь от ошибки первой функции.

В то же время L_1 сообщает что точка данных представляет BMW с вероятностью 99 % и Prius с вероятностью 1 %, а L_2 — что точка данных представляет BMW с вероятностью 99,9 % и Prius с вероятностью 0,1 %. То есть ее прогноз хуже не на 0,9 %, а в десять раз! Следовательно, мы можем считать, что L_2 в десять раз более ошибочна, чем L_1.

Нам нужна такая функция потерь, которая будет возвращать тем большее значение, чем больше $L(x, p)$ *уверена* в неправильном ответе. Для этого можно взять разницу между $L(x, p)$ и неправильным ответом и передать ее функции, которая делает крошечные значения большими. Например, $L_1(x, p)$ вернула 0,99 для Prius, отклонившись на 0,01 единицы от неправильного ответа, а $L_2(x, p)$ вернула 0,999, отклонившись на 0,001 единицы от неправильного ответа. Примером хорошей функции, дающей большое значение для крошечного аргумента, может служить $-\log(x)$, где log — это функция натурального логарифма. Неважно, знаете ли вы, что делает функция $-\log$, важно то, что она возвращает большие числа для небольших входных значений. График функции $-\log(x)$ показан на рис. 15.17.

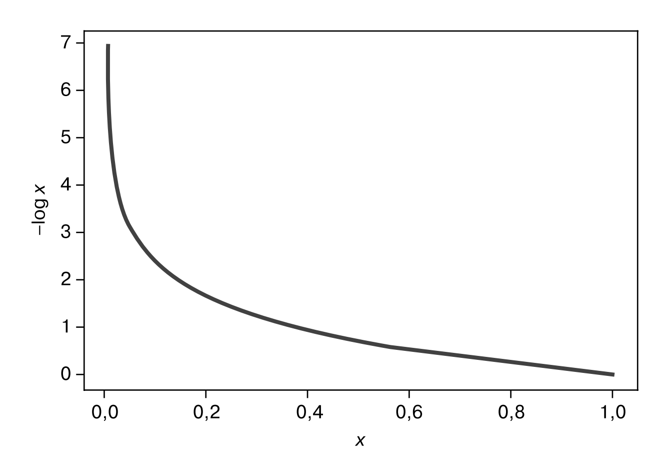


Рис. 15.17. Функция –log (*x*) возвращает большие значения для небольших входных значений, а –log (1) = 0

Для знакомства с функцией $-\log(x)$ можно попробовать применить ее к некоторым небольшим данным. Для $L_1(x, p)$, давшей значение, отстоящее на 0,01 единицы от неправильного ответа, получится меньшая потеря, чем для $L_2(x, p)$, отстоящей на 0,001 единицы от неправильного ответа:

```
from math import log
>>> -log(0.01)
4.605170185988091
>>> -log(0.001)
6.907755278982137
```

Для сравнения: если $L(x, p)$ возвращает ноль для Prius, то это правильный ответ. Этот ответ отстоит от неправильного ответа на одну единицу, а $-\log(1) = 0$, поэтому потери при правильном ответе равны нулю.

Теперь можно реализовать функцию `logistic_cost`, которую мы намеревались создать. Чтобы найти величину потерь для данной точки, вычислим, насколько близко значение данной логистической функции к неправильному ответу, а затем возьмем отрицательный логарифм от результата. Сумма потерь во всех точках данных из набора `scaled_car_data` даст общую величину потерь:

```
def point_cost(l,x,p,is_bmw):        ◄── Определяет величину потерь для одной точки данных
    wrong = 1 - is_bmw
    return -log(abs(wrong - l(x,p)))

def logistic_cost(a,b,c):
    l = make_logistic(a,b,c)
    errors = [point_cost(l,x,p,is_bmw)        ◄── Общая величина потерь логистической функции вычисляется так же как раньше, только теперь для оценки потерь в каждой точке данных используется новая функция point_cost, а не просто абсолютное значение ошибки
              for x,p,is_bmw in scaled_car_data]
    return sum(errors)
```

Попытка минимизировать функцию `logistic_cost` с помощью градиентного спуска дает хороший результат. Но прежде чем убедиться в этом, проверим работоспособность и подтвердим, что `logistic_cost` возвращает более низкие значения для логистической функции с очевидно лучшей границей решения.

15.4.3. Тестирование разных логистических функций

Проверим две логистические функции с разными границами решений и подтвердим, что та из них, которая имеет явно лучшую границу решения, дает меньшее значение потерь. В качестве примеров возьмем $p = 0{,}56 - 0{,}35x$ — мою наилучшую границу решения, которая совпадает с $0{,}35x + 1p = 0{,}56$, а также произвольно выбранную, скажем, $x + p = 1$. Очевидно, что первая точнее отделяет автомобили Prius от BMW.

В примерах с исходным кодом для книги вы найдете функцию `plot_line` для рисования графика прямой по коэффициентам a, b и c из уравнения $ax + by = c$ (в упражнениях в конце раздела вам будет предложено попробовать реализовать ее самостоятельно). Соответствующие значения (a, b, c) равны (0,35, 1, 0,56) и (1, 1, 1). Мы можем нарисовать графики этих линий поверх точек фактических данных (рис. 15.18) всего тремя строками кода:

```
plot_data(scaled_car_data)
plot_line(0.35,1,0.56)
plot_line(1,1,1)
```

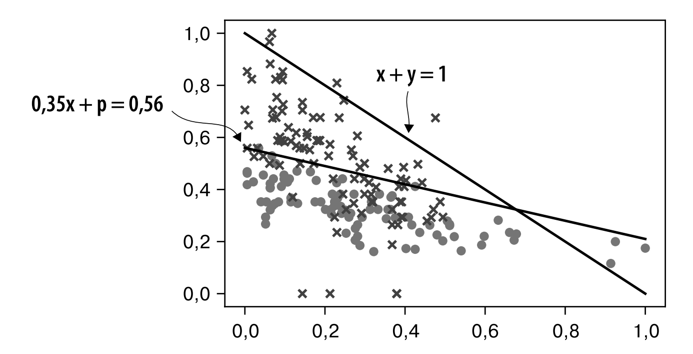


Рис. 15.18. Графики двух границ решения. Один из них явно лучше отделяет автомобили Prius от BMW

Соответствующие логистические функции выглядят так: $\sigma(0{,}35x + p - 0{,}56)$ и $\sigma(x + p - 1)$, и, как мы ожидаем, первая имеет меньшее значение потерь на имеющихся данных. Это можно подтвердить с помощью функции `logistic_cost`:

```
>>> logistic_cost(0.35,1,0.56)
130.92490748700456
>>> logistic_cost(1,1,1)
135.56446830870456
```

Как и ожидалось, прямая $x + p = 1$ — это худшая граница решения из двух, поэтому логистическая функция $\sigma(x + p - 1)$ имеет более высокие потери. Первая функция, $\sigma(0{,}35x + p - 0{,}56)$, имеет меньшее значение потерь и лучше соответствует данным. Но является ли она самой лучшей? Мы узнаем это в следующем разделе, когда используем градиентный спуск с функцией `logistic_cost`.

15.4.4. Упражнения

Упражнение 15.6. Реализуйте функцию `plot_line(a,b,c)`, упомянутую в разделе 15.4.3, которая строит прямую $ax + by = c$, где $0 \le x \le 1$ и $0 \le y \le 1$.

Решение. Обратите внимание на то, что я взял для аргументов функции имена, отличные от a, b и c, потому что `c` — это именованный аргумент, определяющий цвет прямой на графике для функции `plot` из библиотеки Matplotlib, которую я обычно использую:

```
def plot_line(acoeff,bcoeff,ccoeff,**kwargs):
    a,b,c = acoeff, bcoeff, ccoeff
    if b == 0:
        plt.plot([c/a,c/a],[0,1])
    else:
        def y(x):
            return (c-a*x)/b
        plt.plot([0,1],[y(0),y(1)],**kwargs)
```

Упражнение 15.7. Используйте формулу сигмоидной функции и запишите развернутую формулу для $\sigma(ax + by - c)$.

Решение. Учитывая, что

$$\sigma(x) = \frac{1}{1+e^{-x}},$$

можно записать:

$$\sigma(ax+by+c) = \frac{1}{1+e^{c-ax-by}}.$$

Упражнение 15.8. Мини-проект. Как выглядит график $k(x, y) = \sigma(x^2 + y^2 - 1)$? Как выглядит граница решения, то есть множество точек, для которых $k(x, y) = 0{,}5$?

Решение. Мы знаем, что $\sigma(x^2 + y^2 - 1) = 0{,}5$, где $x^2 + y^2 - 1 = 0$ или $x^2 + y^2 = 1$. Решениями этого уравнения можно считать точки, отстоящие на одну единицу от начала координат, то есть окружность с радиусом 1. Внутри окружности расстояние от начала координат меньше единицы, поэтому $x^2 + y^2 < 1$ и $\sigma(x^2 + y^2) < 0{,}5$, а вне окружности $x^2 + y^2 > 1$ и $\sigma(x^2 + y^2 - 1) > 0{,}5$. График этой функции приближается к 1 по мере удаления от начала координат в любом направлении, в то время как внутри окружности стремится к минимальному значению около 0,27 в начале координат. Вот график этой функции.

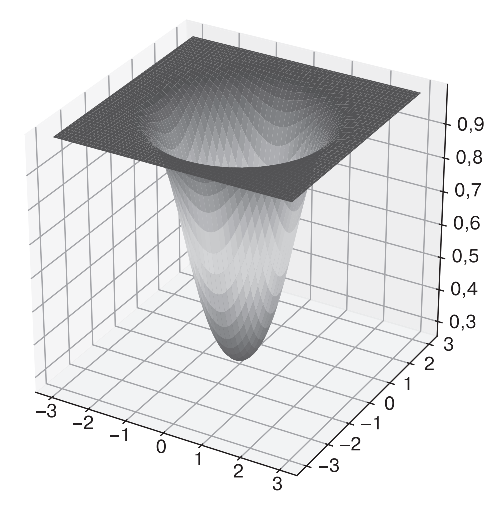


График функции $\sigma(x^2 + y^2 - 1)$. Она дает значение меньше 0,5 внутри окружности с радиусом 1 и увеличивается до 1 при удалении от центра в любом направлении за пределами этого круга

Упражнение 15.9. Мини-проект. Два уравнения, $2x + y = 1$ и $4x + 2y = 2$, определяют одну и ту же прямую и, следовательно, одну и ту же границу решения. Являются ли одинаковыми логистические функции $\sigma(2x + y - 1)$ и $\sigma(4x + 2y - 2)$?

Решение. Нет, это разные функции. Значения $4x + 2y - 2$ растут быстрее с увеличением x и y, поэтому вторая функция имеет более крутой график:

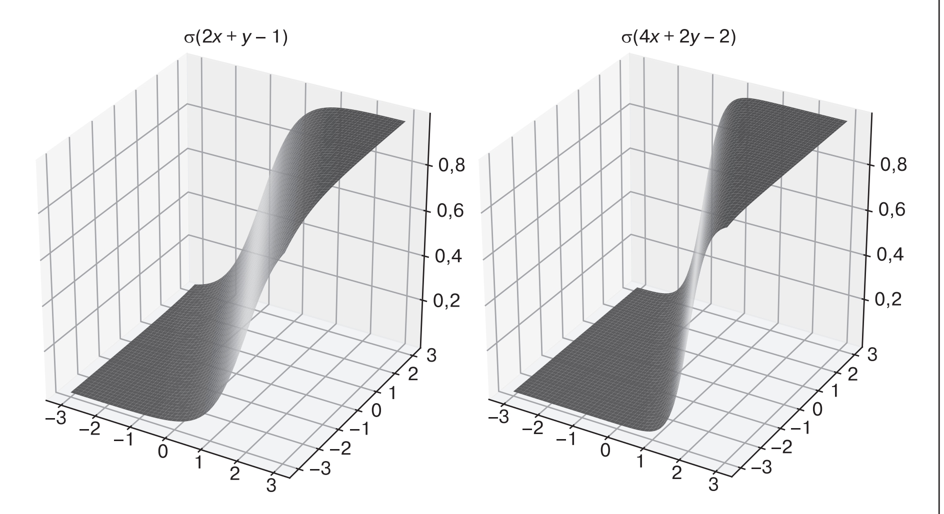


График второй логистической функции круче графика первой

Упражнение 15.10. Мини-проект. Имея уравнение прямой $ax + by = c$, непросто определить, что находится выше, а что ниже этой прямой. Сможете ли вы описать, с какой стороны прямой функция $z(x, y) = ax + by - c$ возвращает положительные значения?

Решение. Прямая $ax + by = c$ представляет множество точек, в которых $z(x, y) = ax + by - c = 0$. Как мы видели в главе 7, график $z(x, y) = ax + by - c$ — это плоскость, поэтому она возрастает в одном направлении от прямой и убывает в другом. Градиент $z(x, y)$ равен $\nabla z(x, y) = (a, b)$, поэтому $z(x, y)$ быстрее всего возрастает в направлении вектора (a, b) и быстрее всего убывает в противоположном направлении — в направлении вектора $(-a, -b)$. Оба эти направления перпендикулярны направлению прямой.

15.5. ПОИСК ЛУЧШЕЙ ЛОГИСТИЧЕСКОЙ ФУНКЦИИ

Теперь мы должны решить простую задачу минимизации — найти значения a, b и c, минимизирующие функцию `logistic_cost`. С этими значениями функция $L(x, p) = \sigma(ax + bp - c)$ будет лучше всего соответствовать данным, и ее можно будет использовать для построения классификатора, принимающего пробег x и цену p неизвестного автомобиля и сообщающего, на какую модель он больше всего похож — BMW, если $L(x, p) > 0{,}5$, или Prius в ином случае. Этот классификатор, назовем его `best_logistic_classifier(x,p)`, можно передать в `test_classifier`, чтобы увидеть, хорошо ли он работает.

Единственное, что мы фактически должны сделать, — обновить нашу функцию `gradient_descent`. До сих пор мы использовали градиентный спуск только с функциями, принимающими двухмерные векторы и возвращающими числа. Функция `logistic_cost` принимает трехмерный вектор (a, b, c) и возвращает число, поэтому нам нужна новая версия градиентного спуска. К счастью, мы уже рассматривали трехмерные аналогии для всех операций с двухмерными векторами, поэтому сделать это будет несложно.

15.5.1. Градиентный спуск в трех измерениях

Еще раз посмотрим, как выполняется расчет градиента для функций двух переменных, который мы использовали в главах 12 и 14. Частные производные функции $f(x, y)$ в точке (x_0, y_0) — это отдельные производные по осям x и y, полученные в предположении, что другая переменная — константа. Например, подставив y_0 на место второго аргумента $f(x, y)$, мы получим $f(x, y_0)$, которую можно рассматривать как функцию только от x и производную которой брать как обычно. Объединение двух частных производных как компонентов двухмерного вектора дает градиент:

```
def approx_gradient(f,x0,y0,dx=1e-6):
    partial_x = approx_derivative(lambda x:f(x,y0),x0,dx=dx)
    partial_y = approx_derivative(lambda y:f(x0,y),y0,dx=dx)
    return (partial_x,partial_y)
```

Отличие функции трех переменных в том, что мы должны взять еще одну частную производную. Если посмотреть на $f(x, y, z)$ в некоторой точке (x_0, y_0, z_0), можно рассматривать $f(x, y_0, z_0)$, $f(x_0, y, z_0)$ и $f(x_0, y_0, z)$ как функции от x, y и z соответственно и взять три частные производные. Объединив их в вектор, мы получим трехмерную версию градиента:

```
def approx_gradient3(f,x0,y0,z0,dx=1e-6):
    partial_x = approx_derivative(lambda x:f(x,y0,z0),x0,dx=dx)
    partial_y = approx_derivative(lambda y:f(x0,y,z0),y0,dx=dx)
    partial_z = approx_derivative(lambda z:f(x0,y0,z),z0,dx=dx)
    return (partial_x,partial_y,partial_z)
```

Процедура градиентного спуска в трех измерениях такая же, как и следовало ожидать. Спуск начинается с некоторой точки в трехмерном пространстве, для которой вычисляется градиент и выполняется небольшой шаг в полученном направлении, чтобы оказаться в новой точке, где, как предполагается, значение $f(x, y, z)$ меньше. В качестве одного из дополнительных улучшений я добавил параметр `max_steps`, чтобы можно было задать максимальное количество шагов градиентного спуска. Если для этого параметра установить разумное ограничение, то нам не придется беспокоиться об остановке программы, даже если алгоритм не сойдется к точке в пределах допуска. Вот как эта процедура выглядит в коде на Python:

```
def gradient_descent3(f,xstart,ystart,zstart,
                      tolerance=1e-6,max_steps=1000):
    x = xstart
    y = ystart
    z = zstart
    grad = approx_gradient3(f,x,y,z)
    steps = 0
    while length(grad) > tolerance and steps < max_steps:
        x -= 0.01 * grad[0]
        y -= 0.01 * grad[1]
        z -= 0.01 * grad[2]
        grad = approx_gradient3(f,x,y,z)
        steps += 1
    return x,y,z
```

Остается только подключить функцию `logistic_cost`, а функция `gradient_descent3` найдет входные данные, минимизирующие ее.

15.5.2. Использование градиентного спуска для поиска наилучшего соответствия

Проявив разумную осторожность, начнем с небольшого значения параметра `max_steps`, например 100:

```
>>> gradient_descent3(logistic_cost,1,1,1,max_steps=100)
(0.21114493546399946, 5.04543972557848, 2.1260122558655405)
```

Если позволить градиентному спуску сделать 200 шагов вместо 100, то можно увидеть, что ему еще есть куда двигаться:

```
>>> gradient_descent3(logistic_cost,1,1,1,max_steps=200)
(0.884571531298388, 6.657543188981642, 2.955057286988365)
```

Напомню, что полученные результаты — это параметры, необходимые для определения логистической функции, а также параметры (a, b, c), определяющие границу решения в форме $ax + bp = c$. Если выполнить градиентный спуск на 100, 200, 300 шагов и т. д. и нарисовать соответствующие прямые с помощью `plot_line`, то мы увидим, как сходится граница решения (рис. 15.19).

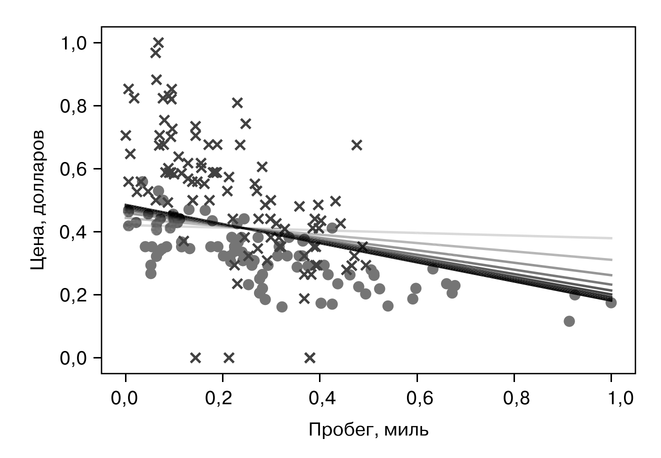


Рис. 15.19. Чем больше шагов, тем ближе значения (*a*, *b*, *c*), возвращаемые градиентным спуском, к параметрам границы решения

Где-то между 7000 и 8000 шагов алгоритм сходится, то есть находит точку, в которой длина градиента меньше 10^{-6}. Собственно, это и есть точка минимума, которую мы ищем:

```
>>> gradient_descent3(logistic_cost,1,1,1,max_steps=8000)
(3.7167003153580045, 11.422062409195114, 5.596878367305919)
```

Сравним эту границу решения с той, которую мы использовали (результат показан на рис. 15.20):

```
plot_data(scaled_car_data)
plot_line(0.35,1,0.56)
plot_line(3.7167003153580045, 11.422062409195114, 5.596878367305919)
```

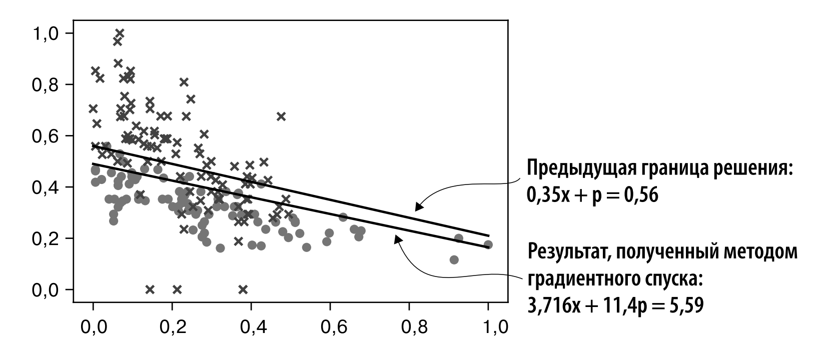


Рис. 15.20. Сравнение предыдущей границы решения с границей, полученной в результате градиентного спуска

Новая граница решения не слишком далека от предыдущей. Как видите, логистическая регрессия сместила границу принятия решения немного вниз, обменяв

несколько ложноположительных классификаций (несколько машин Prius теперь оказались выше прямой, что неверно) на несколько истинно положительных (несколько машин BMW теперь оказались выше прямой, а это правильно).

15.5.3. Оценка лучшего логистического классификатора

Полученные значения (*a*, *b*, *c*) можно подставить в логистическую функцию, а затем использовать ее для создания функции классификации автомобилей:

```
def best_logistic_classifier(x,p):
    l = make_logistic(3.7167003153580045, 11.422062409195114, 5.596878367305919)
    if l(x,p) > 0.5:
        return 1
    else:
        return 0
```

Передав эту функцию в вызов `test_classifier`, мы увидим, что уровень ее точности на наборе тестовых данных примерно равен уровню, полученному в нашей лучшей попытке, — 80 %:

```
>>> test_classifier(best_logistic_classifier,scaled_car_data)
0.8
```

Границы решений довольно близки, поэтому нет ничего странного в том, что качество не слишком изменилось по сравнению с предположениями, сделанными в разделе 15.2. И все же если предыдущая граница давала близкое качество, то почему градиентный спуск сошелся там, где сошелся?

Оказывается, логистическая регрессия не просто ищет оптимальную границу решения. Фактически граница решения, которую мы получили в начале раздела, превосходила этот наилучший логистический классификатор на 0,5 %, поэтому последний не обеспечивает максимальной точности на наборе тестовых данных. Скорее, логистическая регрессия рассматривает набор данных в целом и ищет модель, которая, вероятнее всего, будет точной для всех примеров, а не старается еще немного сдвинуть границу решения, чтобы получить один-два процентных пункта точности. То есть алгоритм ориентирует границу решения, основываясь на целостном представлении набора данных. Если набор данных репрезентативен, то мы можем смело доверять результатам логистического классификатора, полученным на данных, которых он прежде не видел, а не только на данных из обучающего набора.

Другая информация, которую имеет логистический классификатор, — это определенность в отношении каждой точки, которую он классифицирует. Классификатор, основанный только на границе решения, на 100 % уверен, что точка выше этой границы — это BMW, а точка ниже — Prius. Наш логистический классификатор имеет более дифференцированное представление: мы можем

интерпретировать возвращаемые им значения от нуля до единицы как вероятность того, что автомобиль — это модель BMW, а не Prius. Для реальных применений полезно знать не только лучшее предположение, сделанное моделью машинного обучения, но и насколько эта модель заслуживает доверия. Если бы мы классифицировали доброкачественные и злокачественные опухоли на основе результатов медицинских исследований, то могли бы действовать совсем по-другому, если бы алгоритм утверждал, что уверен в злокачественности опухоли на 99 %, а не 51 %.

Уверенность классификатора проявляется в значениях коэффициентов (a, b, c). Например, обратите внимание на то, что соотношение между (a, b, c) в предположении (0,35, 1, 0,56) близко к соотношению найденных оптимальных значений (3,717, 11,42, 5,597). Оптимальные значения примерно в 10 раз превышают наши наилучшие предположения. Это делает график логистической функции более крутым. Оптимальная логистическая функция гораздо более уверена в определении границы решения, чем первая. Это говорит о том, что с пересечением границы решения достоверность результата значительно возрастает, как показано на рис. 15.21.

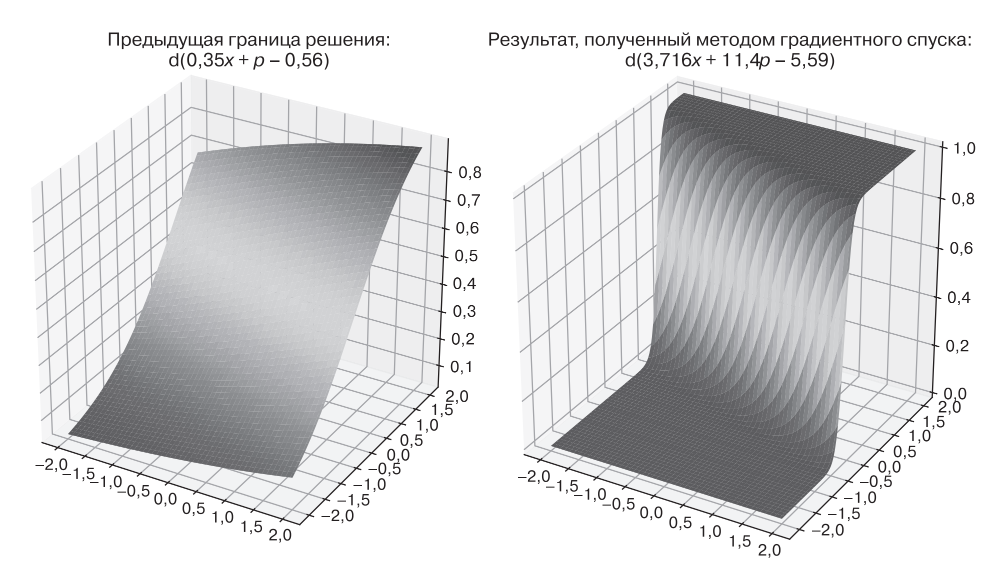


Рис. 15.21. Оптимальная логистическая функция гораздо круче, а это означает, что ее уверенность в выборе между BMW и Prius быстро возрастает по мере отдаления от границы решения

В последней главе мы продолжим применять сигмоидные функции для получения достоверных результатов, находящихся между нулем и единицей, при реализации классификации с использованием нейронных сетей.

15.5.4. Упражнения

Упражнение 15.11. Измените функцию `gradient_descent3` так, чтобы она выводила общее количество выполненных шагов перед возвратом результата. Сколько шагов занимает градиентный спуск, чтобы сойтись для заданной `logistic_cost`?

Решение. Для этого нужно добавить строку `print(steps)` прямо перед возвратом из `gradient_descent3`:

```
def gradient_descent3(f,xstart,ystart,zstart,tolerance=1e-6,max_steps=1000):
    ...
    print(steps)
    return x,y,z
```

Следующая попытка выполнить градиентный спуск

```
gradient_descent3(logistic_cost,1,1,1,max_steps=8000)
```

выведет число `7244`, означающее, что алгоритм сошелся за 7244 шага.

Упражнение 15.12. Мини-проект. Напишите функцию `approx_gradient`, которая вычисляет градиент функции в любом количестве измерений. Затем напишите функцию `gradient_descent`, которая работает в любом количестве измерений. Для проверки своей версии `gradient_descent` на n-мерной функции попробуйте такую функцию, как $f(x_1, x_2, ..., x_n) = (x_1 - 1)^2 + (x_2 - 1)^2 + ... + (x_n - 1)^2$, где $x_1, x_2, ..., x_n$ — n входных переменных функции f. Минимум этой функции должен быть в n-мерной точке $(1, 1, ..., 1)$, все координаты которой равны 1.

Решение. Смоделируем векторы произвольной размерности как списки чисел. Чтобы взять частную производную по i-й координате в векторе $\mathbf{v} = (v_1, v_2, ..., v_n)$, нужно взять обычную производную по i-й координате x_i. То есть мы должны рассматривать функцию

$$f(v_1, v_2, ..., v_{i-1}, x_i, v_{i+1}, ..., v_n).$$

Другими словами, в f подставляются все координаты $\mathbf{v}$, кроме i-го элемента, который остается как переменная x_i. Это дает функцию одной переменной x_i, а ее обычная производная есть i-я частная производная. Вот как выглядит код взятия частных производных:

```
def partial_derivative(f,i,v,**kwargs):
    def cross_section(x):
```

```
        arg = [(vj if j != i else x) for j,vj in enumerate(v)]
        return f(*arg)
    return approx_derivative(cross_section, v[i], **kwargs)
```

Обратите внимание на то, что нумерация координат начинается с нуля, а размерность входных данных для f определяется длиной **v**.

Все остальное реализуется относительно просто. Чтобы построить градиент, мы просто берем n частных производных по порядку и помещаем их в список:

```
def approx_gradient(f,v,dx=1e-6):
    return [partial_derivative(f,i,v) for i in range(0,len(v))]
```

Чтобы выполнить градиентный спуск, заменим все манипуляции с именованными координатными переменными, такими как x, y и z, операциями со списками над вектором координат **v**:

```
def gradient_descent(f,vstart,tolerance=1e-6,max_steps=1000):
    v = vstart
    grad = approx_gradient(f,v)
    steps = 0
    while length(grad) > tolerance and steps < max_steps:
        v = [(vi - 0.01 * dvi) for vi,dvi in zip(v,grad)]
        grad = approx_gradient(f,v)
        steps += 1
    return v
```

Чтобы реализовать предложенную тестовую функцию, можно написать ее обобщенную версию, которая принимает любое количество входных данных и возвращает сумму их квадратов разностей от единицы:

```
def sum_squares(*v):
    return sum([(x-1)**2 for x in v])
```

Значение этой функции не может быть меньше нуля, потому что она вычисляет сумму квадратов, а квадрат не может быть меньше нуля. Нулевое значение получается, только если все элементы входного вектора **v** равны единице, соответственно, этот вектор представляет минимум. Наш градиентный спуск подтверждает это, хотя и с небольшой числовой ошибкой, поэтому все выглядит хорошо! Обратите внимание на то, что начальный вектор **v** пятимерный, поэтому все векторы в вычислениях автоматически являются пятимерными:

```
>>> v = [2,2,2,2,2]
>>> gradient_descent(sum_squares,v)
[1.0000002235452137,
1.0000002235452137,
1.0000002235452137,
1.0000002235452137,
1.0000002235452137]
```

Упражнение 15.13. Мини-проект. Попробуйте выполнить градиентный спуск с функцией потерь `simple_logistic_cost`. Что получится в результате?

Решение. Алгоритм не сходится. Значения a, b и c продолжают неограниченно увеличиваться даже после стабилизации границы решения. Это означает, что по мере того как градиентный спуск исследует все больше и больше логистических функций, они остаются ориентированными в одном и том же направлении, но становятся бесконечно крутыми. У алгоритма сохраняется стимул становиться все ближе и ближе к большинству точек, игнорируя те, которые он распознал неправильно. Как я уже упоминал, эту проблему можно решить, введя штраф за ошибочную классификацию, что и реализовано в функции `logistic_cost`.

КРАТКИЕ ИТОГИ ГЛАВЫ

- Классификация — это тип задач машинного обучения, когда алгоритму предлагается проанализировать немаркированные точки данных и определить, к какому классу принадлежит каждая из них. В этой главе мы рассмотрели данные о пробеге и ценах на подержанные автомобили и написали алгоритм, классифицирующий их либо как BMW 5-й серии, либо как Toyota Prius.
- Простой способ классификации двухмерных векторных данных состоит в том, чтобы установить границу решения, то есть буквально нарисовать в двухмерном пространстве линию, по одну сторону которой будут находиться точки, принадлежащие одному классу, а с другой — точки, принадлежащие другому классу. Простейшая граница решения — это прямая.
- Если граница решения имеет вид $ax + by = c$, то величина $ax + by - c$ будет положительна с одной стороны линии и отрицательна с другой. Мы интерпретировали это значение как меру похожести точки данных на BMW. Положительное значение означает, что точка данных больше похожа на BMW, а отрицательное — на Prius.
- Сигмоидная функция, определяемая, как показано далее, принимает числа от $-\infty$ до ∞ и сжимает их в конечный интервал от нуля до единицы:

$$\sigma(x) = \frac{1}{1+e^{-x}}.$$

- Объединив сигмоидную функцию с функцией $ax + by - c$, мы получили новую функцию $\sigma(ax + by - c)$, которая тоже оценивает похожесть точки данных на BMW, но возвращает значения от нуля до единицы. Функции этого вида называют двухмерными логистическими функциями.

- Значение между нулем и единицей, которое выдает логистический классификатор, можно интерпретировать как степень уверенности в том, что точка данных принадлежит к одному из классов. Например, возвращаемые значения 0,51 или 0,99 означают, что, по мнению модели, соответствующие точки данных представляют BMW, но последний прогноз гораздо более надежный.
- Имея подходящую функцию потерь, штрафующую за уверенность в неправильных классификациях, можно использовать градиентный спуск, чтобы найти наилучшую логистическую функцию. Это будет лучший логистический классификатор для набора данных.

16 Обучение нейронных сетей

В этой главе

- ✓ Классификация изображений рукописных цифр как векторных данных.
- ✓ Разработка нейронной сети, называемой многослойным перцептроном.
- ✓ Использование нейронной сети как векторного преобразования.
- ✓ Обучение нейронной сети на данных с помощью функции потерь и градиентного спуска.
- ✓ Расчет частных производных для нейронных сетей при обратном распространении.

В заключительной главе мы объединим почти все, что узнали, и познакомимся с одним из самых известных инструментов машинного обучения, используемых в настоящее время, — искусственными нейронными сетями. *Искусственные нейронные сети*, или просто нейронные сети, — это математические функции, структура которых в общих чертах основана на структуре человеческого мозга. Их называют искусственными, чтобы отличить от органических нейронных сетей, существующих в мозгу. Нейронные сети могут показаться высокой и сложной целью, но в действительности они основаны на простой метафоре, представляющей работу мозга.

Прежде чем объяснить метафору, я хочу сказать, что я не невролог. В общих чертах суть ее состоит в том, что мозг представляет собой большое скопление

клеток, связанных между собой и называемых *нейронами*. Когда вы обдумываете какую-то мысль, это выражается в изменении электрической активности определенных нейронов. Эту электрическую активность можно увидеть при правильном сканировании мозга, в процессе которого видно, как меняется активность различных его частей (рис. 16.1).

В отличие от миллиардов нейронов в человеческом мозгу нейронные сети, которые мы будем строить с помощью Python, содержат всего несколько десятков нейронов, и степень активности конкретного нейрона представлена одним числом, называемым его *значением активации*. Когда активируется нейрон в мозгу или в искусственной нейронной сети, он может активировать связанные с ним соседние нейроны. Это позволяет одной идее привести к другой, что можно условно рассматривать как творческое мышление.

С математической точки зрения активация нейрона в нейронной сети является функцией численных значений активации соседних нейронов, связанных с ним. Если нейрон связан с четырьмя другими нейронами, имеющими значения активации a_1, a_2, a_3 и a_4, то его активация будет выражаться как некоторая математическая функция от этих четырех значений, скажем, $f(a_1, a_2, a_3, a_4)$.

На рис. 16.2 показана схема, на которой все нейроны изображены кружками. Я заштриховал нейроны по-разному, чтобы показать, что они имеют разные уровни активации, подобно светлым и темным областям на энцефалограмме.

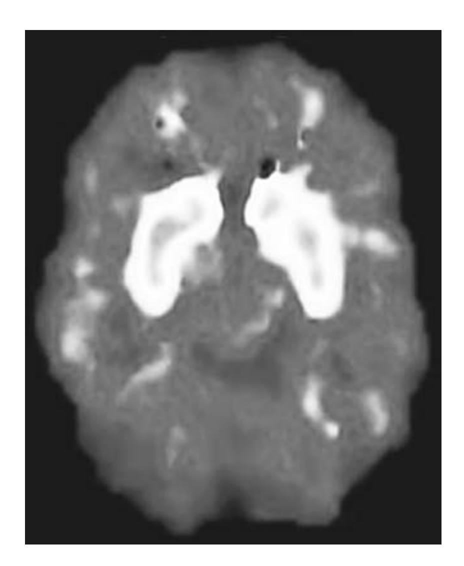


Рис. 16.1. Мозговая деятельность вызывает изменение электрической активности разных нейронов, которое наблюдается в виде появления светлых областей на энцефалограмме

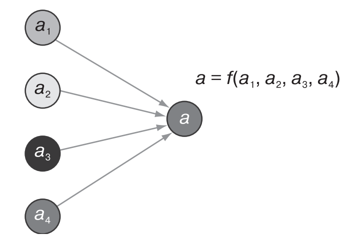


Рис. 16.2. Схема активации нейронов в виде математической функции, где a_1, a_2, a_3 и a_4 — значения активации, подаваемые на вход функции f

Если каждое из значений a_1, a_2, a_3 и a_4 зависит от значения активации других нейронов, то значение a может зависеть от еще большего количества чисел. Имея больше нейронов и связей, можно построить сколь угодно сложную математическую функцию и смоделировать сколь угодно сложные понятия.

Объяснение, которое я только что дал, — несколько отвлеченное философское введение в нейронные сети, и его явно недостаточно, чтобы начать программировать. В этой главе я подробно расскажу, как применять эти концепции и построить собственную нейронную сеть. Так же как в предыдущей главе, мы используем нейронные сети для решения задачи классификации. Создание нейронной сети и ее обучение классификации требует множества шагов, поэтому, прежде чем углубиться в материал, изложу план.

16.1. КЛАССИФИКАЦИЯ ДАННЫХ С ПОМОЩЬЮ НЕЙРОННЫХ СЕТЕЙ

В этом разделе я сосредоточусь на классическом применении нейронных сетей — классификации изображений. В частности, используем изображения рукописных цифр с низким разрешением (числа от 0 до 9) и попробуем обучить нейронную сеть определять, какая цифра представлена на том или ином изображении. На рис. 16.3 показаны примеры изображений цифр.

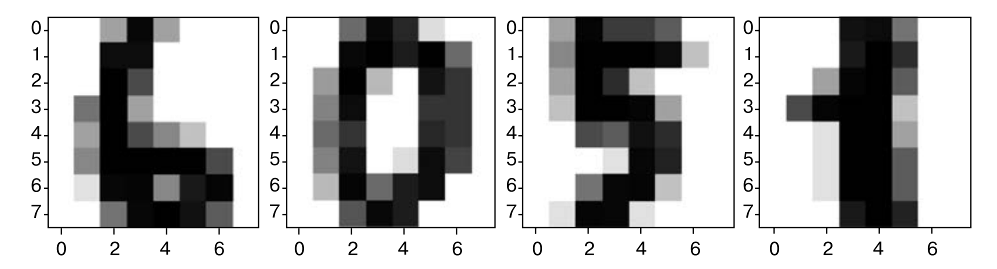


Рис. 16.3. Изображения с низким разрешением некоторых рукописных цифр

Если вы определили цифры на рис. 16.3 как 6, 0, 5 и 1, то поздравляю! Ваша органическая нейронная сеть, то есть мозг, хорошо обучена. Наша цель — построить искусственную нейронную сеть, которая получает такие изображения и классифицирует каждое как одну из десяти возможных цифр, и может быть даже так же хорошо, как это сделал бы человек.

В главе 15 задача классификации сводилась к анализу двухмерного вектора и определению его принадлежности к одному из двух классов. В этой главе будем анализировать черно-белые изображения размером 8×8 пикселов, в которых каждый из 64 пикселов описывается одним числом, определяющим его яркость. По аналогии с тем, как мы рассматривали изображения в виде векторов в главе 6, будем рассматривать 64-пиксельные значения яркости как 64-мерные векторы.

Мы поместим каждый 64-мерный вектор в один из десяти классов, в соответствии с представляемой им цифрой. Таким образом, функция классификации будет иметь больше входных и выходных данных, чем функция в главе 15.

В частности, функция классификации на основе нейронной сети, которую мы создадим, будет выглядеть как функция с 64 аргументами и 10 выходными значениями. Другими словами, она будет выполнять нелинейное векторное преобразование из $\mathbb{R}^{64}$ в $\mathbb{R}^{10}$. Входные значения, масштабированные в диапазоне от 0 до 1, будут представлять насыщенность цвета пикселов, а десять выходных значений — вероятности принадлежности изображения к каждому из десяти классов цифр. Индекс наибольшего выходного значения будет считаться ответом. В примере, изображенном на рис. 16.4, нейронной сети передается изображение цифры 5, а сеть возвращает наибольшее значение в пятой позиции в выходном векторе, правильно идентифицировав цифру на изображении.

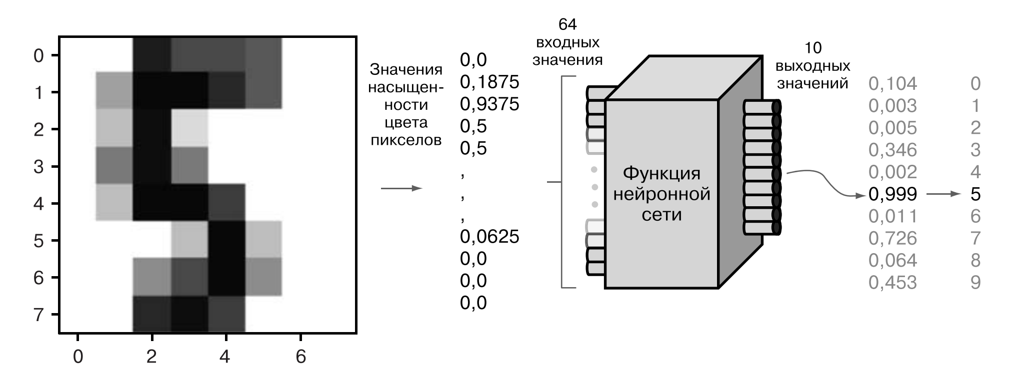


Рис. 16.4. Пример классификации изображения цифры функцией нейронной сети на Python

Функция нейронной сети в центре на рис. 16.4 — не что иное, как математическая функция. Она имеет более сложную структуру, чем функции, которые мы видели до сих пор, и определяющая ее формула слишком длинная, чтобы записать ее на бумаге. Вычисление результата нейронной сети больше похоже на выполнение алгоритма. Я покажу, как это сделать и реализовать на Python.

Так же как в предыдущей главе, где оценивали множество различных логистических функций, мы могли бы опробовать множество разных нейронных сетей и выяснить, какая из них имеет наибольшую точность предсказания. Сделать это можно было бы с помощью градиентного спуска. Однако, в отличие от линейной функции, которая определяется двумя константами, a и b, в формуле $f(x) = ax + b$, нейронная сеть заданной формы может определяться тысячами констант и для градиентного спуска потребуется взять очень много частных производных! К счастью, благодаря форме функций, связывающих нейроны в нейронной сети, для получения градиента можно использовать упрощенный алгоритм, который называется *обратным распространением*.

Мы могли бы вывести алгоритм обратного распространения с нуля и реализовать его с помощью только математических выкладок, которые рассмотрели к настоящему моменту, но, к сожалению, это слишком большой проект для данной книги. Поэтому я покажу, как использовать известную библиотеку scikit-learn (приставка sci происходит от science — научная) на Python для выполнения градиентного спуска, чтобы автоматически обучить нейронную сеть прогнозировать с максимальной точностью. Наконец, я кратко познакомлю вас с математикой, лежащей в основе обратного распространения ошибки, и надеюсь, что это станет отправной точкой для вашей плодотворной карьеры в машинном обучении.

16.2. КЛАССИФИКАЦИЯ ИЗОБРАЖЕНИЙ РУКОПИСНЫХ ЦИФР

Прежде чем приступить к реализации нейронной сети, нужно подготовить данные. Цифровые изображения, которые мы будем использовать, входят в комплект бесплатных тестовых данных, поставляемых вместе с библиотекой scikit-learn. После загрузки нужно будет преобразовать их в 64-мерные векторы со значениями, масштабированными в диапазоне от нуля до единицы. Набор данных содержит также правильные ответы для всех изображений, представленные в виде целых чисел от нуля до девяти.

Затем мы напишем две функции на Python, чтобы попрактиковаться в классификации. Первая — фиктивная функция идентификации цифр `random_classifier`, которая принимает 64 числа, представляющих изображение, и возвращает 10 случайных чисел, выражающих уверенность в том, что изображение представляет цифры от 0 до 9. Вторая — это функция `test_digit_classify`, которая принимает классификатор, автоматически применяет его к каждому изображению в наборе и возвращает количество правильных ответов. Поскольку классификатор `random_classifier` выдает случайные результаты, он должен угадывать правильный ответ только в 10 % случаев. Это обеспечит основу для дальнейшего улучшения при замене фиктивного классификатора настоящей нейронной сетью.

16.2.1. Построение 64-мерных векторов изображения

Если вы используете дистрибутив Anacondas Python, как предлагается в приложении A, то библиотека scikit-learn уже должна быть доступна как `sklearn`. В противном случае можете установить ее с помощью `pip`. Чтобы открыть `sklearn` и импортировать набор данных с цифрами, вам понадобится выполнить следующий код:

```
from sklearn import datasets
digits = datasets.load_digits()
```

Каждый элемент набора представлен двухмерным массивом NumPy (матрицей), задающим значения пикселов для одного изображения. Например, `digits.images[0]` задает значения пикселов первого изображения в наборе данных — матрицу 8 × 8:

```
>>> digits.images[0]
array([[ 0., 0., 5., 13., 9., 1., 0., 0.],
       [ 0., 0., 13., 15., 10., 15., 5., 0.],
       [ 0., 3., 15., 2., 0., 11., 8., 0.],
       [ 0., 4., 12., 0., 0., 8., 8., 0.],
       [ 0., 5., 8., 0., 0., 9., 8., 0.],
       [ 0., 4., 11., 0., 1., 12., 7., 0.],
       [ 0., 2., 14., 5., 10., 12., 0., 0.],
       [ 0., 0., 6., 13., 10., 0., 0., 0.]])
```

Здесь видно, что диапазон значений оттенков серого ограничен. Матрица содержит только целые числа от 0 до 15.

В библиотеке Matplotlib имеется удобная функция `imshow`, которая показывает элементы матрицы в виде изображения. При правильной спецификации оттенков серого нули в матрице отображаются как белые квадратики, а большие ненулевые значения — как квадратики более темного оттенка серого. Например, на рис. 16.5 показано первое изображение из набора данных, похожее на цифру 0, полученное с помощью `imshow`:

```
import matplotlib.pyplot as plt
plt.imshow(digits.images[0], cmap=plt.cm.gray_r)
```

Чтобы еще раз подчеркнуть, что мы будем интерпретировать изображения как 64-мерные векторы, на рис. 16.6 показана версия изображения с каждым из 64 значений яркости, наложенных на соответствующие пикселы.

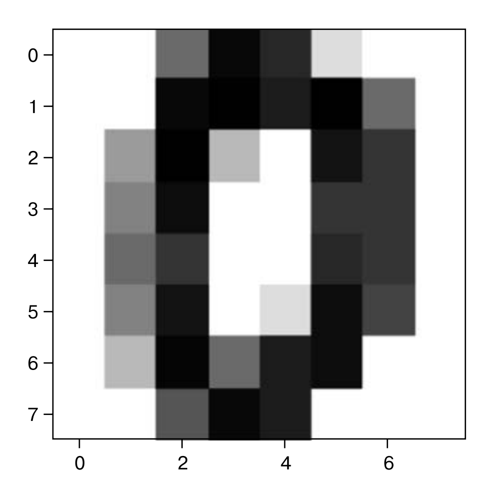


Рис. 16.5. Первое изображение из набора данных sklearn, похожее на цифру 0

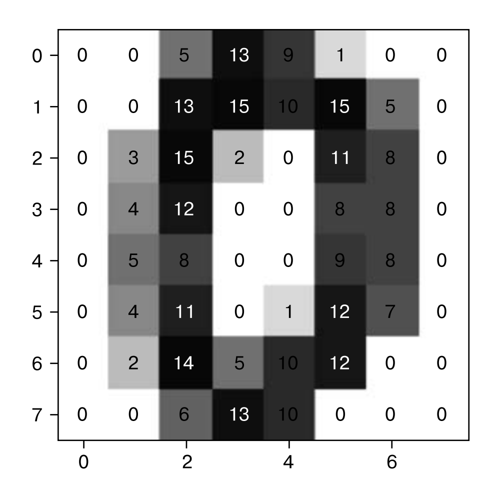


Рис. 16.6. Изображение из набора данных со значениями яркости для каждого пиксела

Превратить матрицу чисел 8×8 в один вектор с 64 элементами можно с помощью функции `np.matrix.flatten` из библиотеки NumPy. Эта функция конструирует вектор, начиная с первой строки матрицы, за которой следует вторая строка и т. д., что позволяет получить векторное представление изображения, аналогичное использованному в главе 6. Преобразование первой матрицы с изображением действительно дает вектор с 64 элементами:

```
>>> import numpy as np
>>> np.matrix.flatten(digits.images[0])
array([ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.,  0.,  0., 13., 15., 10.,
       15.,  5.,  0.,  0.,  3., 15.,  2.,  0., 11.,  8.,  0.,  0.,  4.,
       12.,  0.,  0.,  8.,  8.,  0.,  0.,  5.,  8.,  0.,  0.,  9.,  8.,
        0.,  0.,  4., 11.,  0.,  1., 12.,  7.,  0.,  0.,  2., 14.,  5.,
       10., 12.,  0.,  0.,  0.,  0.,  6., 13., 10.,  0.,  0.,  0.])
```

Чтобы предотвратить появление вычислительных ошибок, мы вновь прибегнем к масштабированию данных, чтобы значения находились в диапазоне от 0 до 1. Поскольку все значения пикселов для каждой записи в этом наборе данных находятся в диапазоне от 0 до 15, можно умножить эти векторы на скаляр 1/15, чтобы получить масштабированные версии.

NumPy предоставляет перегруженные версии операторов `*` и `/` для автоматического умножения и деления векторов на скаляр, поэтому мы можем просто ввести инструкцию

```
np.matrix.flatten(digits.images[0]) / 15
```

и получить масштабированный результат. Теперь перейдем к созданию примера классификатора цифр, которому можно передать эти значения.

16.2.2. Построение случайного классификатора цифр

На вход классификатора цифр передается 64-мерный вектор, подобный тем, которые мы только что построили, а на выходе возвращается 10-мерный вектор со значениями в диапазоне от 0 до 1. В нашем первом примере элементы выходного вектора генерируются случайно, но мы интерпретируем их как уверенность классификатора в том, что изображение представляет каждую из 10 цифр.

Поскольку генерирование случайных чисел — не проблема, мы легко можем реализовать создание выходного вектора. В библиотеке NumPy есть функция `np.random.rand`, которая создает массив случайных чисел от 0 до 1 заданного размера. Например, `np.random.rand(10)` создаст массив с 10 случайными числами в диапазоне от 0 до 1. Функция `random_classifier` принимает входной вектор, игнорирует его и возвращает случайно сгенерированный вектор:

```
def random_classifier(input_vector):
    return np.random.rand(10)
```

Вот как можно выполнить классификацию первого изображения в наборе данных:

```
>>> v = np.matrix.flatten(digits.images[0]) / 15.
>>> result = random_classifier(v)
>>> result
array([0.78426486, 0.42120868, 0.47890909, 0.53200335, 0.91508751,
       0.1227552, 0.73501115, 0.71711834, 0.38744159, 0.73556909])
```

Наибольший элемент в этом векторе равен `0.915` и имеет индекс 4. Возвращая этот вектор, классификатор сообщает о своей уверенности в том, что, скорее всего, изображение представляет цифру 4. Чтобы получить индекс максимального значения программно, можно использовать такой код на Python:

```
>>> list(result).index(max(result))
4
```

Здесь `max(result)` отыскивает наибольший элемент в массиве, а `list(result)` обрабатывает массив как обычный список Python. Для поиска индекса элемента можно использовать встроенную функцию `index` списка. Возвращаемое значение 4 неверно — мы уже видели, что изображение представляет цифру 0, и то же самое говорит официальный результат.

Правильная цифра для каждого изображения хранится в элементе с соответствующим индексом в массиве `digits.target`. Для изображения `digits.images[0]` правильным значением будет `digits.target[0]`, которое равно нулю, как мы и ожидали:

```
>>> digits.target[0]
0
```

Случайный классификатор предсказал, что изображение представляет цифру 4, тогда как на самом деле это цифра 0. Поскольку классификатор генерирует прогнозы случайным образом, он должен ошибаться в 90 % случаев, и мы можем подтвердить это, выполнив проверку на множестве примеров.

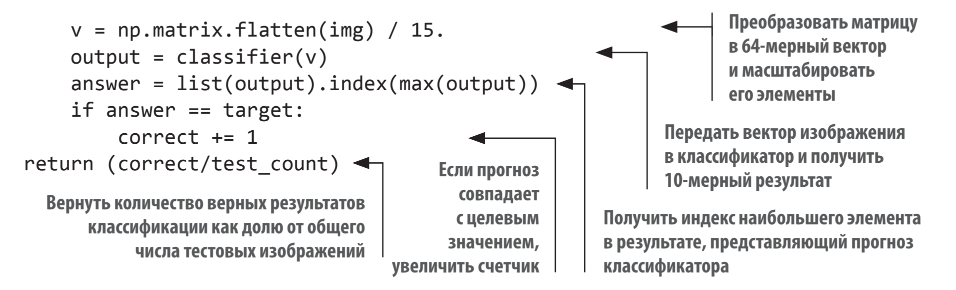
16.2.3. Оценка характеристик классификатора цифр

Теперь напишем функцию `test_digit_classify`, которая принимает функцию классификатора и оценивает ее качество на большом наборе изображений цифр. Любая функция классификатора будет иметь одну и ту же форму — принимать 64-мерный вектор на входе и возвращать 10-мерный вектор на выходе. Функция `test_digit_classify` проверяет все тестовые изображения и известные правильные ответы и сообщает количество правильных ответов, полученных от классификатора:

```
def test_digit_classify(classifier,test_count=1000):
    correct = 0
    for img, target in zip(digits.images[:test_count],
                           digits.target[:test_count]):
```

Первоначально счетчик верных ответов равен нулю

Цикл по парам изображений и целевых значений в тестовом наборе данных



Ожидается, что случайный классификатор даст правильные ответы примерно в 10 % случаев. Действуя случайным образом, в некоторых испытаниях он может работать лучше, в некоторых хуже, но так как мы тестируем очень много изображений, то результат каждый раз должен быть близким к 10 %. Давайте попробуем:

```
>>> test_digit_classify(random_classifier)
0.107
```

В этом испытании наш случайный классификатор показал себя чуть лучше, чем ожидалось, дав верные ответы в 10,7 % случаев. Сам по себе этот результат не особенно интересен, зато теперь у нас есть организованные данные и базовый пример, который нужно превзойти, чтобы начать строить нейронную сеть.

16.2.4. Упражнения

Упражнение 16.1. Предположим, что функция классификатора возвращает такой массив NumPy:

```
array([5.00512567e-06, 3.94168539e-05, 5.57124430e-09, 9.31981207e-09,
       9.98060276e-01, 9.10328786e-07, 1.56262695e-03, 1.82976466e-04,
       1.48519455e-04, 2.54354113e-07])
```

Какую цифру, по его мнению, представляет изображение?

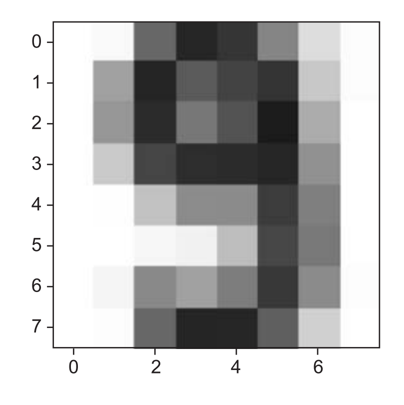
Решение. Наибольший элемент в данном массиве имеет значение 9,98060276e–01, или примерно 0,998. Это пятый элемент, то есть имеющий индекс 4. Отсюда следует, что, по мнению классификатора, изображение представляет цифру 4.

Упражнение 16.2. Мини-проект. Найдите среднее для всех изображений девяток в наборе данных подобно тому, как мы вычисляли средние значения изображений в главе 6. Выведите полученное изображение. На что оно похоже?

Решение. Этот код принимает целое число i и усредняет изображения в наборе данных, представляющие цифру i. Поскольку изображения цифр представлены в виде массивов NumPy, которые поддерживают сложение и умножение на скаляр, мы можем усреднить их, используя обычную функцию `sum` и оператор деления:

```
def average_img(i):
    imgs = [img for img,target in zip(digits.images[1000:],
                                      digits.target[1000:]) if target==i]
    return sum(imgs) / len(imgs)
```

Вызов `average_img(9)` вычислит матрицу 8×8, представляющую среднее всех изображений девяток. Вот как оно выглядит.



Упражнение 16.3. Мини-проект. Создайте классификатор, превосходящий случайный, который вычислял бы среднее изображение для каждой цифры в тестовом наборе данных и сравнивал классифицируемое изображение со всеми средними значениями. В частности, классификатор должен возвращать вектор скалярных произведений классифицируемого изображения на усредненные изображения цифр.

Решение

```
avg_digits = [np.matrix.flatten(average_img(i)) for i in range(10)]
def compare_to_avg(v):
    return [np.dot(v,avg_digits[i]) for i in range(10)]
```

Оценка этого классификатора показывает, что он верно классифицирует 85 % изображений цифр. Неплохо!

```
>>> test_digit_classify(compare_to_avg)
0.853
```

16.3. ПРОЕКТИРОВАНИЕ НЕЙРОННОЙ СЕТИ

В этом разделе я покажу, что нейронная сеть по сути является математической функцией, и расскажу, какого поведения можно ожидать в зависимости от ее структуры. Это подготовит нас к следующему разделу, где мы реализуем свою первую нейронную сеть как функцию на Python, которая будет классифицировать изображения цифр.

Для задачи классификации изображений наша нейронная сеть будет принимать на входе 64 значения, выдавать на выходе 10 значений и выполнять сотни операций. По этой причине здесь я покажу в качестве примера более простую сеть с тремя входами и двумя выходами. Это позволит представить всю сеть и пройти каждый этап ее вычислений. После знакомства с этим примером вам будет проще реализовать вычислительные этапы для нейронной сети любого размера.

16.3.1. Организация нейронов и связей между ними

Как я писал в начале этой главы, нейронная сеть — это совокупность нейронов, в которой каждый конкретный нейрон активируется в зависимости от степени активности связанных с ним нейронов. Математически степень активности нейрона — это функция активностей связанных с ним нейронов. Поведение нейронной сети может различаться в зависимости от количества используемых нейронов, связей между ними и описывающих их функций. В этой главе ограничимся одним из самых простых видов нейронных сетей — многослойным перцептроном.

Многослойный перцептрон (multilayer perceptron, MLP) состоит из нескольких столбцов нейронов, называемых *слоями*, расположенных в порядке слева направо. Активация каждого нейрона является функцией активации в предыдущем слое, то есть в слое, находящемся непосредственно слева от него. Крайний левый слой не зависит ни от каких других нейронов, и его активация основана на обучающих данных. На рис. 16.7 показана схема четырехслойного MLP.

На рис. 16.7 каждый кружок — это нейрон, а линии между кружками представляют связи между нейронами. Активация нейрона зависит только от активации нейронов в предыдущем слое и влияет на активацию всех нейронов в следующем слое. Количество нейронов в каждом слое я выбрал произвольно, и на этой конкретной схеме слои состоят из трех, четырех, трех и двух нейронов.

Поскольку в данном примере всего 12 нейронов, мы имеем 12 значений активации. Часто используются сети с намного бо́льшим числом нейронов (сеть,

предназначенная для классификации цифр, будет содержать 90 нейронов), поэтому у нас не получится определить переменную для каждого из них. Вместо этого мы обозначим все активации буквой «*a*» и применим верхние и нижние индексы. Верхний индекс будет определять слой, а нижний — порядковый номер нейрона в нем. Например, a_2^2 — это число, представляющее активацию второго нейрона во втором слое.

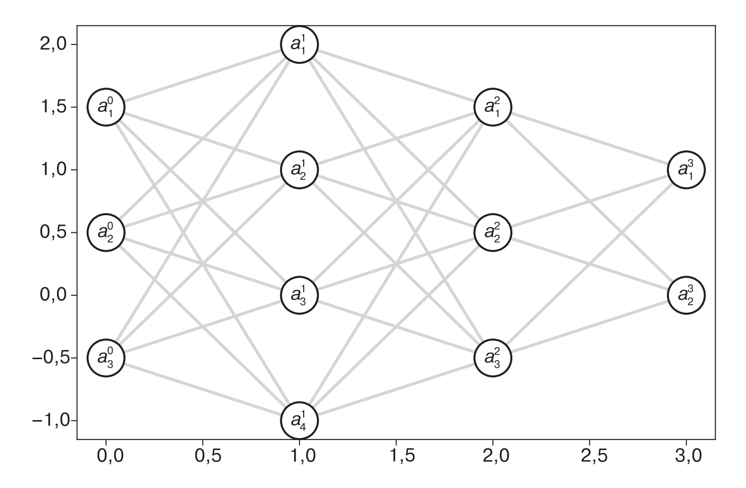


Рис. 16.7. MLP, состоящий из нескольких слоев нейронов

16.3.2. Поток данных через нейронную сеть

Чтобы вычислить результат нейронной сети как математической функции, нужно выполнить три основных шага, которые я опишу далее в терминах значений активации. Сначала я представлю основную суть, а затем покажу формулы. Помните, нейронная сеть — это просто функция, которая принимает вектор на входе и создает вектор на выходе. Шаги между ними — это просто рецепт получения выходного вектора из заданного входного вектора. Вот первый шаг в конвейере.

Шаг 1: настройка активаций входного слоя значениями /из входного вектора

Входной слой — это другое название первого или самого левого слоя. Сеть на рис. 16.7 имеет три нейрона во входном слое, поэтому может принимать трехмерные векторы. Если предположить, что входной вектор состоит из значений (0,3, 0,9, 0,5), то мы можем выполнить первый шаг, установив $a_1^0 = 0{,}3$, $a_2^0 = 0{,}9$ и $a_3^0 = 0{,}5$. Этот шаг заполняет 3 из 12 нейронов сети (рис. 16.8).

Каждое значение активации в первом слое является значением функции активации в нулевом слое. Теперь у нас достаточно информации для их расчета, и можно переходить к шагу 2.

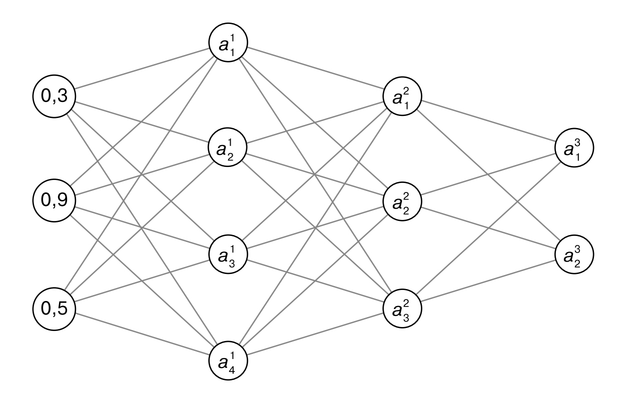


Рис. 16.8. Установка активаций нейронов входного слоя равными элементам входного вектора (*слева*)

Шаг 2: вычисление каждой активации в следующем слое как функцию всех активаций во входном слое

Этот шаг — основа вычислений, и я снова вернусь к нему, когда закончу концептуальный обзор всех шагов. Сейчас важно понять, что каждая активация в следующем слое обычно задается *отдельной функцией* в предыдущем слое. Скажем, мы хотим вычислить a_0^1. Эта активация является некоторой функцией a_1^0, a_2^0 и a_3^0, которую можем записать как $a_1^1 = f(a_1^0, a_2^0, a_3^0)$. Предположим, мы вычисляем $f(0{,}3,\ 0{,}9,\ 0{,}5)$ и ответ равен 0,6. Тогда значение a_1^1 станет равным 0,6 (рис. 16.9).

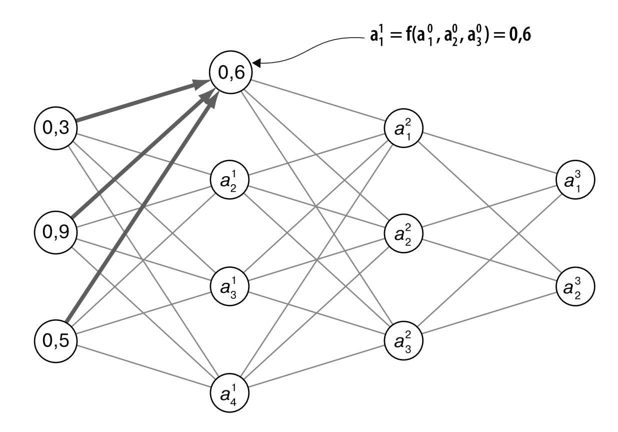


Рис. 16.9. Вычисление активации в первом слое как функции активаций в нулевом слое

Следующая активация в первом слое, a_2^1, тоже вычисляется как функция активаций a_1^0, a_2^0 и a_3^0, но в целом это другая функция, скажем, $a_2^1 = g(a_1^0, a_2^0, a_3^0)$. Результат по-прежнему зависит от тех же входных данных, но является результатом другой функции, которая, вероятно, дает другой результат. Допустим, $g(0{,}3, 0{,}9, 0{,}5) = 0{,}1$, и это значение станет значением активации для a_2^1 (рис. 16.10).

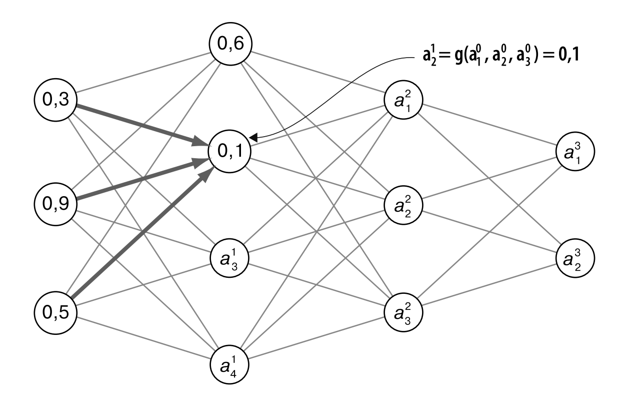


Рис. 16.10. Вычисление другой активации в первом слое как другой функции активаций в нулевом слое

Я использовал f и g в качестве простых имен функций. Кроме них есть две отдельные функции для вычисления активаций a_3^1 и a_4^1. Я не буду давать имена этим функциям, потому что буквы быстро закончатся, однако важно понять, что каждая активация вычисляется как особая функция активаций предыдущего слоя. Вычислив все активации нейронов в первом слое, мы заполним 7 из 12 активаций. Результат может выглядеть примерно так, как показано на рис. 16.11.

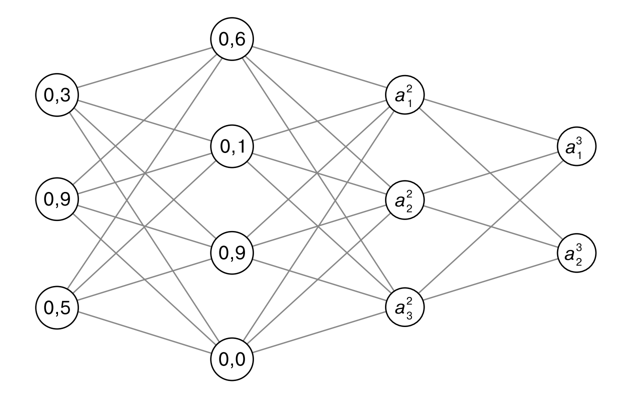


Рис. 16.11. Результаты вычислений активаций в двух слоях нашего MLP

Далее процесс повторяется, пока не будут вычислены значения активаций всех остальных нейронов в сети, и это шаг 3.

Шаг 3: повторение процесса вычисления активаций для каждого следующего слоя на основе активаций предыдущего слоя

Мы начинаем с вычисления a_1^2 как функции активаций первого слоя, a_1^1, a_2^1, a_3^1 и a_4^1. Затем переходим к a_2^2 и a_3^2, активации которых задаются их собственными функциями. Наконец, вычисляем активации a_1^3 и a_2^3, используя их собственные функции активаций второго слоя. Активации для всех нейронов в сети вычислены (рис. 16.12).

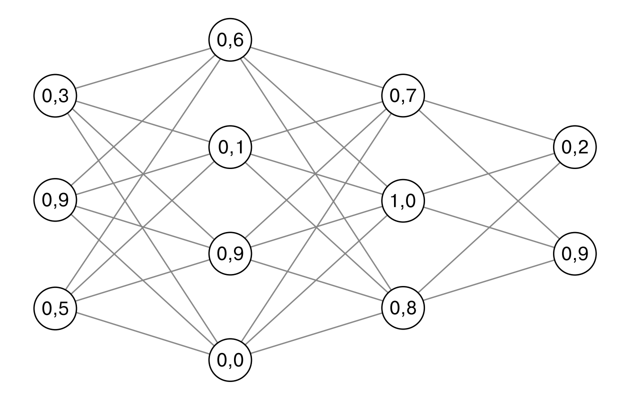


Рис. 16.12. Пример MLP со всеми активациями

На этом вычисления заканчиваются. Мы получили активации для слоев, расположенных в середине, которые называются *скрытыми слоями*, и для последнего слоя, называемого *выходным слоем*. Теперь осталось лишь прочитать активации выходного слоя, чтобы получить результат, и это шаг 4.

Шаг 4: вернуть вектор, элементы которого являются активациями выходного слоя

В этом примере на выходе получается вектор (0,2, 0,9), то есть результатом нашей нейронной сети как функции входного вектора (0,3, 0,9, 0,5) является выходной вектор (0,2, 0,9).

Вот и все! Единственное, что я пропустил, — как рассчитать отдельные активации, и это то, что отличает нейронные сети. У каждого нейрона кроме связей с нейронами из предыдущего слоя есть своя функция, и параметры, определяющие ее, — это числа, которые можно настраивать, чтобы заставить нейронную сеть делать то, что нам нужно.

16.3.3. Вычисление активаций

Самое интересное, что для вычисления активаций в слое мы будем использовать знакомые нам логистические функции. Сложность заключается в том, что нейронная сеть имеет девять нейронов за пределами входного слоя, поэтому требуется

следить за параметрами девяти различных функций. Более того, существует несколько констант, определяющих поведение каждой логистической функции. Большая часть работы будет заключаться в отслеживании всех этих констант.

Сосредоточимся на конкретном примере. Ранее я уже отметил, что в примере MLP имеются активации, зависящие от трех активаций во входном слое: a_1^0, a_2^0 и a_3^0. Функция, дающая a_1^1, — линейная функция этих входных данных (включая константу), которая сама передается сигмоидной функции. Здесь мы имеем четыре свободных параметра, которые я назову A, B, C и D (рис. 16.13).

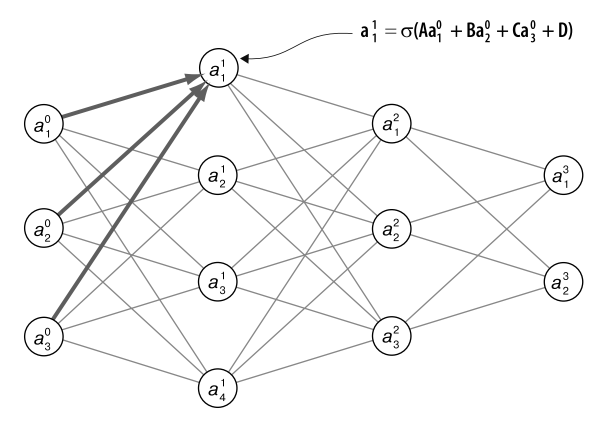


Рис. 16.13. Общий вид функции для вычисления a_1^1 из активаций входного слоя

Нужно настроить переменные A, B, C и D так, чтобы a_1^1 правильно реагировала на входные данные. В главе 15 мы рассматривали логистические функции, принимающие нескольких чисел и возвращающие решение в виде числа от нуля до единицы, измеряющего уверенность в ответе «да». Следуя этой аналогии, нейроны в середине сети можно рассматривать как промежуточные решения более мелких частей общей задачи классификации.

Для каждой связи в сети существует константа, сообщающая, насколько сильно активация входного нейрона влияет на активацию выходного нейрона. В этом случае константа A говорит, насколько сильно на a_1^1 влияет a_1^0, а B и C — насколько сильно на a_1^1 влияют a_2^0 и a_3^0 соответственно. Эти константы называются *весами* нейронной сети, и для каждой связи на общей диаграмме нейронной сети, рассматриваемой в этой главе, определен свой вес.

Константа D не влияет на связь, а просто независимо ни от чего увеличивает или уменьшает значение a_1^1. Эта константа называется *предвзятостью* (*смещенностью*) нейрона, потому что определяет его склонность принимать решение без учета каких-либо входных данных. Слово «*предвзятость*» иногда имеет негативный оттенок, но это важная часть любого процесса принятия решений, так как помогает избежать бессмысленных решений в отсутствие веских доказательств.

Эти объяснения могут показаться запутанными, но как бы то ни было, нам нужно проиндексировать эти веса и смещения и избавиться от конкретных имен, таких как A, B, C и D. Будем обозначать веса как w_{ij}^{l}, где l — номер слоя справа от связи, i — индекс предыдущего нейрона в слое $l-1$, а j — индекс целевого нейрона в слое l. Например, вес A, который воздействует на первый нейрон первого слоя на основе значения первого нейрона нулевого слоя, обозначается w_{11}^{1}. Вес связи, соединяющей второй нейрон третьего слоя с первым нейроном предыдущего слоя, обозначается w_{21}^{3} (рис. 16.14).

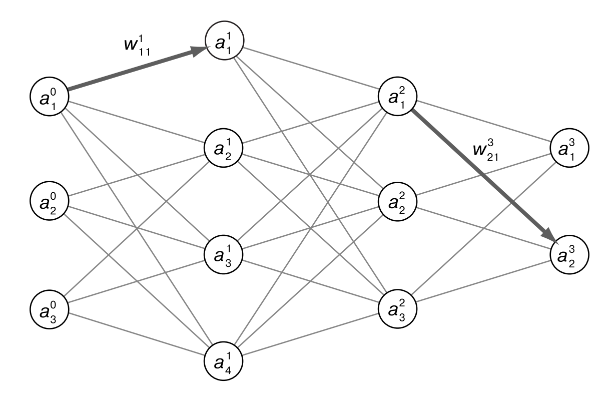


Рис. 16.14. Выделены связи, соответствующие весам w_{11}^1 и w_{21}^3

Предвзятость (смещенность) — это характеристика одного нейрона, а не их пары, поэтому для каждого нейрона существует одно значение смещенности: b_j^l обозначает смещенность j-го нейрона в l-м слое. В терминах этих соглашений об именах мы могли бы записать формулу для a_1^1 как

$$a_1^1 = \sigma\left(w_{11}^1 a_1^0 + w_{12}^2 a_2^0 + w_{13}^3 a_3^0 + b_1^1\right),$$

а формулу для a_3^2 как

$$a_3^2 = \sigma\left(w_{31}^2 a_1^1 + w_{32}^2 a_2^1 + w_{33}^2 a_3^1 + w_{34}^2 a_4^1 + b_3^2\right).$$

Как видите, вычисление активаций не представляет сложности, но количество переменных может сделать этот процесс утомительным и склонным к ошибкам. К счастью, этот процесс можно упростить, использовав матрицы, которые рассматривались в главе 5.

16.3.4. Вычисление активаций в матричной записи

Как бы утомительно это ни было, рассмотрим конкретный пример и напишем формулу активаций для целого слоя сети, а потом посмотрим, как упростить ее с помощью матричной записи, и напишем более практичную формулу. Возьмем второй слой. Вот как выглядят формулы трех активаций:

$$a_1^2 = \sigma\left(w_{11}^2 a_1^1 + w_{12}^2 a_2^1 + w_{13}^2 a_3^1 + w_{14}^2 a_4^1 + b_1^2\right);$$
$$a_2^2 = \sigma\left(w_{21}^2 a_1^1 + w_{22}^2 a_2^1 + w_{23}^2 a_3^1 + w_{24}^2 a_4^1 + b_2^2\right);$$
$$a_3^2 = \sigma\left(w_{31}^2 a_1^1 + w_{32}^2 a_2^1 + w_{33}^2 a_3^1 + w_{34}^2 a_4^1 + b_3^2\right).$$

Полезно именовать величины внутри сигмоидной функции. Обозначим три величины z_1^2, z_2^2 и z_3^2, так что по определению

$$a_1^2 = \sigma\left(z_1^2\right);$$
$$a_2^2 = \sigma\left(z_2^2\right);$$
$$a_3^2 = \sigma\left(z_3^2\right).$$

Формулы для этих значений *z* выглядят проще, потому что являются линейными комбинациями активаций предыдущего слоя плюс константа. Это означает, что можно записать их в матрично-векторной форме, начиная с

$$z_1^2 = w_{11}^2 a_1^1 + w_{12}^2 a_2^1 + w_{13}^2 a_3^1 + w_{14}^2 a_4^1 + b_1^2;$$
$$z_2^2 = w_{21}^2 a_1^1 + w_{22}^2 a_2^1 + w_{23}^2 a_3^1 + w_{24}^2 a_4^1 + b_2^2;$$
$$z_3^2 = w_{31}^2 a_1^1 + w_{32}^2 a_2^1 + w_{33}^2 a_3^1 + w_{34}^2 a_4^1 + b_3^2.$$

Эти три уравнения можно записать как вектор:

$$\begin{pmatrix} z_1^2 \\ z_2^2 \\ z_3^2 \end{pmatrix} = \begin{pmatrix} w_{11}^2 a_1^1 + w_{12}^2 a_2^1 + w_{13}^2 a_3^1 + w_{14}^2 a_4^1 + b_1^2 \\ w_{21}^2 a_1^1 + w_{22}^2 a_2^1 + w_{23}^2 a_3^1 + w_{24}^2 a_4^1 + b_2^2 \\ w_{31}^2 a_1^1 + w_{32}^2 a_2^1 + w_{33}^2 a_3^1 + w_{34}^2 a_4^1 + b_3^2 \end{pmatrix}$$

и затем вынести за скобки вектор смещений:

$$\begin{pmatrix} z_1^2 \\ z_2^2 \\ z_3^2 \end{pmatrix} = \begin{pmatrix} w_{11}^2 a_1^1 + w_{12}^2 a_2^1 + w_{13}^2 a_3^1 + w_{14}^2 a_4^1 \\ w_{21}^2 a_1^1 + w_{22}^2 a_2^1 + w_{23}^2 a_3^1 + w_{24}^2 a_4^1 \\ w_{31}^2 a_1^1 + w_{32}^2 a_2^1 + w_{33}^2 a_3^1 + w_{34}^2 a_4^1 \end{pmatrix} + \begin{pmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \end{pmatrix}.$$

Это простое сложение трехмерных векторов. Несмотря на то что большой вектор в середине выглядит как большая матрица, это всего лишь вектор-столбец из трех сумм. Однако этот большой вектор можно разложить на произведение матрицы на вектор:

$$\begin{pmatrix} z_1^2 \\ z_2^2 \\ z_3^2 \end{pmatrix} = \begin{pmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 & w_{14}^2 \\ w_{21}^2 & w_{22}^2 & w_{23}^2 & w_{24}^2 \\ w_{31}^2 & w_{32}^2 & w_{33}^2 & w_{34}^2 \end{pmatrix} \begin{pmatrix} a_1^1 \\ a_2^1 \\ a_3^1 \\ a_4^1 \end{pmatrix} + \begin{pmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \end{pmatrix}.$$

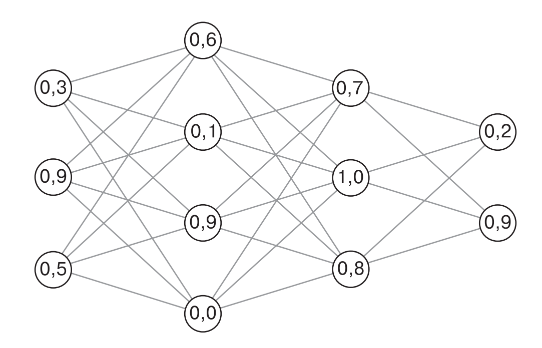
Далее вычисляются активации второго слоя путем применения σ к каждому элементу получившегося вектора. Это всего лишь упрощение обозначений, но

психологически полезно выделить числа w^l_{ij} и b^l_j в отдельные матрицы. Эти числа определяют саму нейронную сеть, в отличие от активаций a^l_j, которые являются промежуточными шагами в вычислениях.

Чтобы понять, что я имею в виду, сравните вычисления в нейронной сети с вычислением функции $f(x) = ax + b$. Входная переменная — x, а a и b — это константы, определяющие функцию: пространство возможных линейных функций определяется выбором a и b. Величина ax, даже если обозначить ее как q, — это просто промежуточный шаг в вычислении $f(x)$. Согласно этой аналогии, после того как вы определите количество нейронов на слой в своем многослойном перцептроне, матрица весов и векторы смещений для каждого слоя будут фактически определять нейронную сеть. С учетом всего этого можно реализовать MLP на Python.

16.3.5. Упражнения

Упражнение 16.4. Какой нейрон и в каком слое представлен активацией a^3_2? Какое значение имеет эта активация на следующем изображении? (Нейроны и слои пронумерованы так же, как в предыдущих разделах.)



Решение. Верхний индекс определяет слой, а нижний индекс — нейрон внутри слоя. Соответственно, активация a^3_2 представляет второй нейрон в слое 3. На изображении она имеет значение 0,9.

Упражнение 16.5. Если слой 5 нейронной сети содержит 10 нейронов, а слой 6 — 12 нейронов, сколько всего связей будет образовано между нейронами 5-го и 6-го слоев?

Решение. Каждый из 10 нейронов слоя 5 связан с каждым из 12 нейронов слоя 6. Всего 120 связей.

Упражнение 16.6. Пусть имеется MLP с 12 слоями. Какие индексы l, i и j будет иметь вес связи w_{ij}^{l} между третьим нейроном 4-го слоя и седьмым нейроном 5-го слоя?

Решение. Напомню, что l — это номер целевого слоя связи, поэтому в данном случае $l = 5$. Индексы i и j относятся к нейронам в слоях l и $l - 1$ соответственно, поэтому $i = 7$ и $j = 3$. Соответственно, вес будет обозначаться w_{73}^{5}.

Упражнение 16.7. Где находится вес w_{31}^{3} в сети, используемой как пример в этом разделе?

Решение. Такого веса нет. Этот конкретный вес описывает связь с третьим нейроном в третьем (выходном) слое, но в этом слое у нас только два нейрона.

Упражнение 16.8. Напишите формулу a_1^3 для нейронной сети из этого раздела с точки зрения активаций слоя 2, а также весов и смещений.

Решение. Активациями предыдущего слоя являются a_1^2, a_2^2 и a_3^2, а веса связей, соединяющие их с a_1^3, обозначаются как w_{11}^3, w_{12}^3 и w_{13}^3. Смещение для активации a_1^3 обозначается b_1^3, поэтому формула записывается так:

$$a_1^3 = \sigma\left(w_{11}^3 a_1^2 + w_{12}^3 a_2^2 + w_{13}^3 a_3^2 + b_1^3\right).$$

Упражнение 16.9. Мини-проект. Напишите на Python функцию `sketch_mlp(*layer_sizes)`, которая принимает размеры слоев нейронной сети и выводит диаграмму, подобную той, что использовалась в иллюстрациях в этом разделе. Все нейроны должны быть подписаны соответствующими метками и связаны прямыми отрезками, иллюстрирующими связи. Вызов `sketch_mlp(3,4,3,2)` должен нарисовать схему, которую мы задействовали для представления нейронной сети в этом разделе.

Решение. Исходный код функции вы найдете в примерах исходного кода для книги.

16.4. СОЗДАНИЕ НЕЙРОННОЙ СЕТИ НА PYTHON

Здесь я покажу, как использовать процедуру вычислений MLP, описанную в предыдущем разделе, и реализовать ее на Python. В частности, мы создадим класс `MLP`, который хранит веса и смещения, сгенерированные случайным образом, и предоставляет метод `evaluate`, принимающий 64-мерный и возвращающий 10-мерный вектор. Этот код является простым механистическим переводом на Python архитектуры MLP, описанной в предыдущем разделе. Но как только закончим реализацию, вы сможете протестировать его на задаче классификации рукописных цифр.

Пока веса и смещения выбираются случайным образом, он, скорее всего, мало чем будет отличаться от случайного классификатора, созданного нами в начале главы. Но имея структуру нейронной сети, мы можем настроить веса и смещения и повысить ее прогнозирующую способность. Впрочем, эту задачу решим в следующем разделе.

16.4.1. Реализация класса MLP на Python

Чтобы наш класс представлял многослойный перцептрон, нужно указать количество слоев и количество нейронов в каждом из них. Для инициализации перцептрона с желаемой структурой конструктор может принимать список чисел, определяющих количество нейронов в каждом слое.

Для вычислений понадобятся веса и смещения для каждого слоя, следующего за входным. Как мы только что видели, веса можно хранить в виде матрицы (массив NumPy), а смещения — в виде вектора (тоже массив NumPy). Для начала используем случайные значения весов и смещений, а в процессе обучения сети будем постепенно заменять их более значимыми.

Кратко пройдемся по размерам весовых матриц и векторов со смещениями, которые нам понадобятся. Если текущий слой имеет m нейронов, а предыдущий — n нейронов, то веса будут описывать линейную часть преобразования n-мерного вектора активаций в m-мерный вектор. Для этого нужна матрица размером $m \times n$, то есть матрица с m строками и n столбцами. Чтобы убедиться в этом, вернемся к примеру из раздела 16.3, где веса связей, соединяющих слой с четырьмя нейронами со слоем с тремя нейронами, составляют матрицу 4×3, как показано на рис. 16.15.

Вектор смещений для слоя с m нейронами содержит m элементов, по одному для каждого нейрона. Теперь, определив, как найти размер матрицы весов и векторы смещений для каждого слоя, можно реализовать их создание в конструкторе класса. Обратите внимание на то, что итерации начинаются с элемента `layer_sizes[1:]`, так мы пропускаем первый — входной слой:

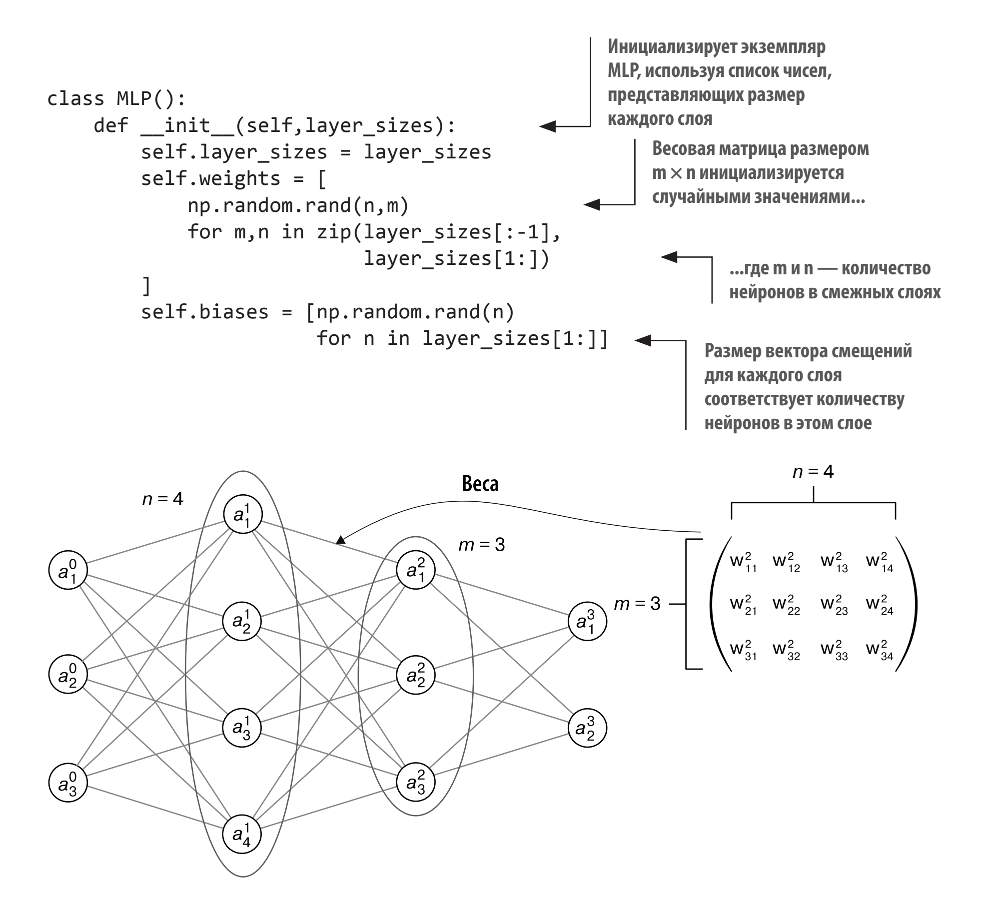


Рис. 16.15. Матрица весов связей, соединяющих слой с четырьмя нейронами со слоем с тремя нейронами, — это матрица 3×4

Теперь убедимся, что при создании экземпляра двухслойного перцептрона конструктор создает ровно одну весовую матрицу и один вектор смещений соответствующих размеров. Пусть первый слой состоит из двух нейронов, а второй — из трех. Выполним этот код:

```
>>> nn = MLP([2,3])
>>> nn.weights
[array([[0.45390063, 0.02891635],
        [0.15418494, 0.70165829],
        [0.88135556, 0.50607624]])]
>>> nn.biases
[array([0.08668222, 0.35470513, 0.98076987])]
```

Результаты подтверждают, что для этого перцептрона была создана одна весовая матрица 3×2 и один трехмерный вектор смещений, заполненные случайными элементами.

Количество нейронов во входном и выходном слоях должно соответствовать размерам векторов, которые передаются на вход и получаются на выходе. В задаче классификации изображений на вход будут подаваться 64-мерные векторы, а на выходе возвращаться 10-мерные. Исходные условия диктуют, что входной слой должен содержать 64 нейрона, а выходной — 10 нейронов, однако я добавлю еще один промежуточный слой с 16 нейронами. Выбор правильного количества слоев и их размеров, чтобы нейронная сеть хорошо справлялась с поставленной задачей, — не столько наука, сколько искусство, и за это специалистам по машинному обучению платят большие деньги. В данном случае я утверждаю, что этой структуры вполне достаточно, чтобы получить модель, обладающую хорошей прогностической силой.

Нашу нейронную сеть можно инициализировать вызовом `MLP([64,16,10])`, и она намного больше любой из тех, что мы рисовали до сих пор. На рис. 16.16 показано, как она выглядит.

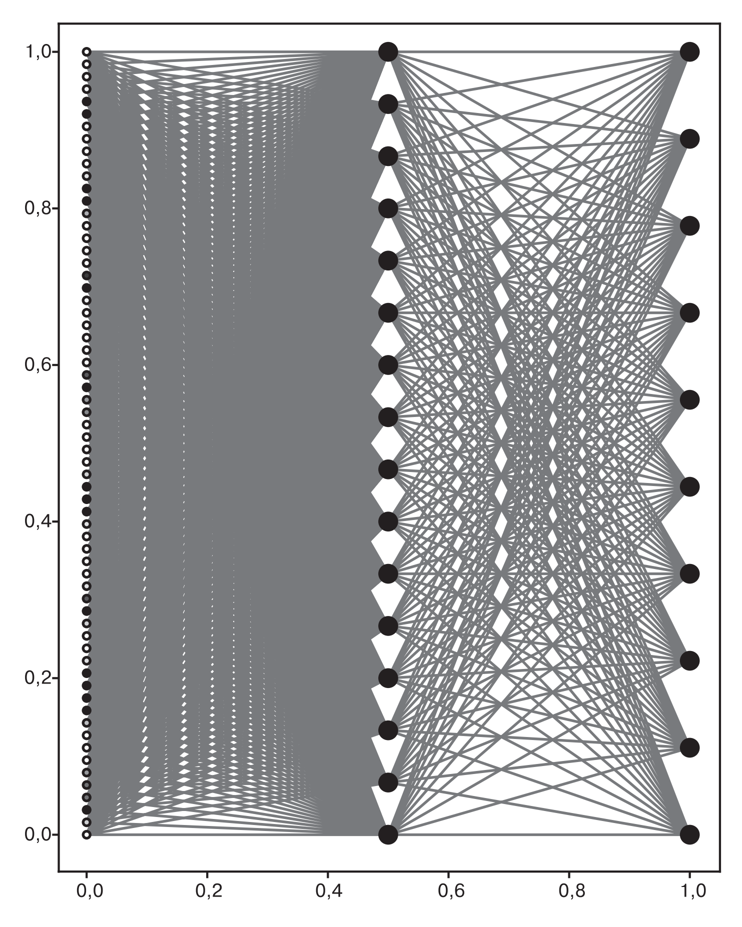


Рис. 16.16. Три слоя MLP с 64, 16 и 10 нейронами

К счастью, как только мы реализуем метод `evaluate`, выполнить вычисления в большой сети будет не сложнее, чем в маленькой, потому что всю работу сделает Python!

16.4.2. Вычисления в MLP

Метод вычислений для нашего класса MLP должен принимать 64-мерный и возвращать 10-мерный вектор. Процедура вычислений от входа до выхода основана на послойной обработке активаций от входного слоя до выходного. Как вы увидите далее, когда мы будем обсуждать обратное распространение, все активации должны сохраняться по мере продвижения, даже для скрытых слоев в середине сети. По этой причине я создам функцию `evaluate` в два этапа: сначала создам метод для вычисления всех активаций, а затем добавлю еще один — для получения значений активаций последнего слоя и вычисления результатов.

Первый метод я назову `feedforward`. Это типичное имя процедуры для послойного вычисления активаций. Активации входного слоя задаются входным вектором, и, чтобы перейти к следующему слою, нужно умножить вектор этих активаций на весовую матрицу, прибавить вектор смещений следующего слоя и пропустить результаты через сигмоидную функцию. Этот процесс будет повторяться, пока мы не доберемся до выходного слоя. Вот как это выглядит:

```
class MLP():
    ...
    def feedforward(self,v):
        activations = []
        a = v
        activations.append(a)
        for w,b in zip(self.weights, self.biases):
            z = w @ a + b
            a = [sigmoid(x) for x in z]
            activations.append(a)
        return activations
```


Активации последнего слоя — это нужные нам результаты, поэтому метод `evaluate` для нейронной сети просто вызывает метод `feedforward` для входного вектора, а затем извлекает последний вектор активаций:

```
class MLP():
    ...
    def evaluate(self,v):
        return np.array(self.feedforward(v)[-1])
```

Вот и все! Как видите, матричное умножение избавило нас от множества циклов перебора нейронов, которые в противном случае пришлось бы написать для вычисления активаций.

16.4.3. Проверка качества классификации моделью MLP

Теперь модель MLP соответствующего размера может принимать вектор с изображением цифры и выводить результат:

```
>>> nn = MLP([64,16,10])
>>> v = np.matrix.flatten(digits.images[0]) / 15.
>>> nn.evaluate(v)
array([0.99990572, 0.9987683 , 0.99994929, 0.99978464, 0.99989691,
       0.99983505, 0.99991699, 0.99931011, 0.99988506, 0.99939445])
```

Здесь мы передали на вход 64-мерный вектор, представляющий изображение, и получили 10-мерный вектор на выходе, то есть нейронная сеть правильно выполняет векторное преобразование. Поскольку веса и смещения инициализированы случайными значениями, результат не может быть хорошим предсказанием цифры на изображении. (Между прочим, все числа близки к 1, потому что все веса, смещения и входные числа положительны, а сигмоида преобразует большие положительные числа в значения, близкие к 1.) Несмотря на это в выходном векторе есть *самый большой* элемент — число с индексом 2. Оно неверно предсказывает, что изображение 0 в наборе данных представляет число 2.

Случайность предполагает, что наш перцептрон правильно угадывает только 10 % ответов. Мы можем подтвердить это с помощью функции `test_digit_classify`. Для только что созданного случайного MLP она дала ровно 10 %:

```
>>> test_digit_classify(nn.evaluate)
0.1
```

На первый взгляд мы не достигли никакого прогресса и все же можем смело похвалить себя, потому что получили работоспособный классификатор, даже при том что он не особенно хорошо справляется со своей задачей. Вычисления в нейронной сети гораздо сложнее, чем вычисление простой функции, такой как $f(x) = ax + b$, но вскоре, когда обучим нейронную сеть более точно классифицировать изображения, мы увидим отдачу.

16.4.4. Упражнения

Упражнение 16.10. Мини-проект. Перепишите метод `feedforward`, использовав явные циклы по слоям и весам, без матричного умножения. Убедитесь, что полученный результат точно совпадает с предыдущей реализацией.

16.5. ОБУЧЕНИЕ НЕЙРОННОЙ СЕТИ С ПОМОЩЬЮ ГРАДИЕНТНОГО СПУСКА

Обучение нейронной сети может показаться абстрактной идеей, однако в действительности под этим понимается всего лишь поиск наилучших весов и смещений, которые заставят нейронную сеть решать поставленную задачу как можно лучше. Мы не сможем охватить здесь весь алгоритм, но я объясню, как он работает на концептуальном уровне и как использовать стороннюю библиотеку, чтобы выполнить обучение автоматически. К концу раздела мы настроим веса и смещения нейронной сети так, что она с высокой степенью точности будет предсказывать, какая цифра представлена на изображении. После этого снова проверим ее с помощью `test_digit_classify` и оценим, насколько хорошо она работает.

16.5.1. Обучение как задача минимизации

В предыдущих главах, где обсуждались линейная функция $ax + b$ и логистическая функция $\sigma(ax + by + c)$, мы создали функцию потерь, которая измеряла несоответствие линейной или логистической функции фактическим данным в зависимости от констант в формуле. Константами в линейной функции были наклон и точка пересечения с осью y — a и b, поэтому функция потерь имела форму $C(a, b)$. Логистическая функция определялась константами a, b и c, поэтому ее функция потерь имела вид $C(a, b, c)$. Обе эти функции потерь зависели от всех обучающих примеров. Чтобы найти лучшие параметры, мы будем использовать градиентный спуск и с его помощью минимизируем функцию потерь.

Самое большое отличие MLP состоит в том, что его поведение может зависеть от сотен и даже тысяч констант: всех его весов w_{ij}^l и смещений b_j^l для каждого слоя l и действительных индексов нейронов i и j. Наша нейронная сеть с 64, 16 и 10 нейронами в трех слоях имеет $64 \cdot 16 = 1024$ веса между первыми двумя слоями и $16 \cdot 10 = 160$ весов между последними. Она также имеет 16 смещений в скрытом слое и 10 смещений в выходном. Всего нужно настроить 1210 констант. Попробуйте представить функцию потерь, как функцию этих 1210 значений, которые нужно минимизировать. Если попробовать записать ее, она будет выглядеть примерно так:

$$C\left(w_{11}^1, w_{12}^1, \ldots, b_1^1, b_2^1, \ldots\right).$$

Под многоточиями в этом уравнении подразумеваются еще более 1000 весов и 24 смещения. Стоит немного подумать о том, как создать функцию потерь. Попробуйте сделать это самостоятельно в качестве упражнения.

Наша нейронная сеть выдает векторы, но мы считаем, что ответом в задаче классификации является цифра на изображении. Чтобы решить эту проблему,

представим правильный ответ, как 10-мерный вектор, который идеальный классификатор будет давать на выходе. Например, если изображение четко представляет цифру 5, мы хотели бы увидеть 100%-ную уверенность в том, что изображение представляет 5, и 0%-ную уверенность в том, что любую другую цифру. То есть выходной вектор должен иметь 1 в элементе с индексом 5 и 0 в других элементах (рис. 16.17).

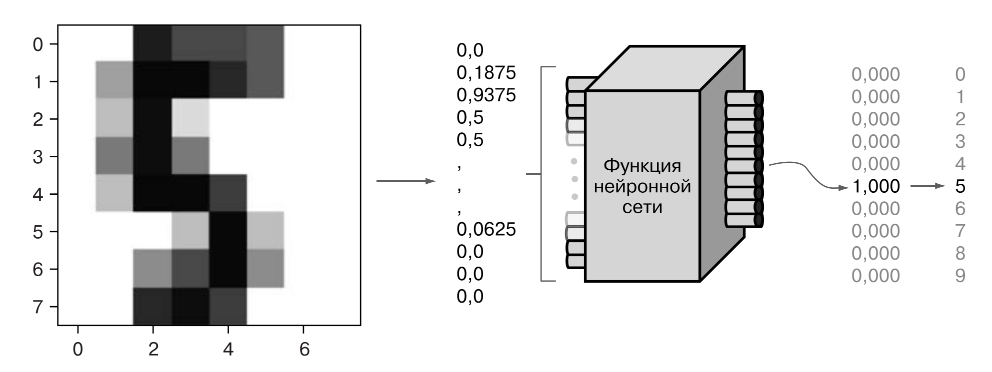


Рис. 16.17. Идеальный результат нейронной сети: 1,0 в элементе с правильным индексом и 0,0 в других элементах

Наша нейронная сеть, как и предыдущие попытки регрессии, никогда не будет в точности соответствовать данным. Чтобы измерить ошибку 10-мерного выходного вектора, можно вычислить квадрат расстояния в 10 измерениях между идеальным и реально полученным векторами.

Предположим, что идеальный вектор записывается как $y = (y_1, y_2, y_3, ..., y_{10})$. Обратите внимание на то, что здесь я следую математическому соглашению об индексации с 1, а не принятому в Python соглашению об индексации с 0. Это же соглашение использовалось для нейронов внутри слоя, поэтому активации выходного слоя (второй слой) индексируются как $\left(a_1^2, a_2^2, a_3^2, \ldots, a_{10}^2\right)$. Квадрат расстояния между этими векторами равен сумме

$$\left(y_1 - a_1^2\right)^2 + \left(y_2 - a_2^2\right)^2 + \left(y_3 - a_3^2\right)^2 + \ldots + \left(y_{10} - a_{10}^2\right)^2.$$

Еще один потенциальный источник путаницы: надстрочный индекс 2 над значениями a указывает, что выходной слой — второй в нашей сети, а 2 за скобками означает возведение в квадрат. Чтобы получить общее значение потерь относительно набора данных, можно вычислить прогнозы нейронной сети для всех образцов изображений и взять среднее квадратическое расстояние. В конце раздела вам представится возможность реализовать это на Python в качестве упражнения.

16.5.2. Вычисление градиентов с обратным распространением

Имея функцию потерь $C\left(w_{11}^1, w_{12}^1, \ldots, b_1^1, b_2^1, \ldots\right)$, реализованную на Python, мы могли бы написать 1210-мерную версию градиентного спуска. Это означало бы использование 1210 частных производных на каждом шаге для получения градиента. Такой градиент будет иметь вид 1210-мерного вектора частных производных в точке:

$$\nabla C\left(w_{11}^1, w_{12}^1, \ldots, b_1^1, b_2^1, \ldots\right) = \left(\frac{\partial C}{\partial w_{11}^1}, \frac{\partial C}{\partial w_{12}^1}, \ldots, \frac{\partial C}{\partial b_1^1}, \frac{\partial C}{\partial b_2^1}, \ldots\right).$$

Оценка такого количества частных производных потребовала бы значительных вычислительных ресурсов, потому что для каждой из них нужно было бы дважды вычислить C, чтобы проверить эффект от настройки одной из входных переменных. В свою очередь оценка C требует просмотра каждого изображения в обучающем наборе и его передачи через нейронную сеть. Может быть, это и можно сделать, но продолжительность вычислений для большинства реальных задач, таких как наша, окажется непомерно большой.

Однако есть более эффективный способ вычисления частных производных — найти их точные формулы, используя методы, подобные рассмотренным в главе 10. Я не буду подробно объяснять, как это сделать, но в последнем разделе дам подсказку. Суть в том, что все 1210 частных производных имеют вид

$$\frac{\partial C}{w_{ij}^l} \text{ или } \frac{\partial C}{b_j^l}$$

для некоторого набора индексов l, i и j. Алгоритм *обратного распространения* вычисляет все эти частные производные рекурсивно, выполняя обход весов и смещений в обратном направлении — от выходного слоя к первому.

Если вам интересно узнать больше об обратном распространении, потерпите до последнего раздела главы. А пока я обращусь к библиотеке scikit-learn, чтобы с ее помощью вычислить потери и выполнить обратное распространение и градиентный спуск.

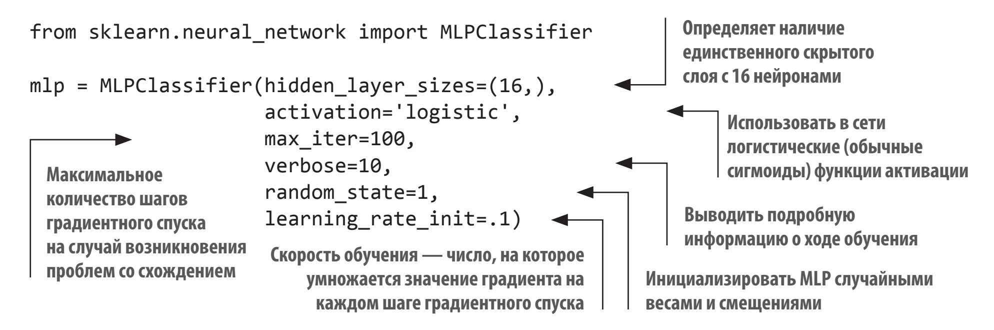
16.5.3. Автоматическое обучение с помощью scikit-learn

Для обучения MLP с помощью scikit-learn не понадобятся никакие новые концепции. Мы можем просто сообщить библиотеке, как настроить задачу, а затем получить ответ. Я не буду объяснять все, на что способна библиотека scikit-learn, но покажу вам код, реализующий обучение MLP классификации цифр.

Первый шаг — поместить все обучающие данные — в нашем случае изображения цифр в виде 64-мерных векторов — в один массив NumPy. Использовав первую 1000 изображений из набора данных, мы получим матрицу 1000 × 64. Поместим также первые 1000 ответов в выходной список:

```
x = np.array([np.matrix.flatten(img) for img in digits.images[:1000]]) / 15.0
y = digits.target[:1000]
```

Затем используем класс `MLP`, входящий в состав scikit-learn, для инициализации перцептрона. Размеры входного и выходного слоев определяются данными, поэтому нам остается только указать размер единственного скрытого слоя посередине. Кроме того, настроим параметры, управляющие обучением MLP. Вот код:



Теперь можно запустить обучение нейронной сети на входных данных x и соответствующих выходных данных y:

```
mlp.fit(x,y)
```

После запуска этой строки кода в процессе обучения нейронной сети будет выводиться информация о его ходе. Эти сведения помогут определить, сколько шагов градиентного спуска необходимо и каково значение функции потерь на каждом шаге:

```
Iteration 1, loss = 2.21958598
Iteration 2, loss = 1.56912978
Iteration 3, loss = 0.98970277
...
Iteration 58, loss = 0.00336792
Iteration 59, loss = 0.00330330
Iteration 60, loss = 0.00321734
Training loss did not improve more than tol=0.000100 for two consecutive
epochs. Stopping.
```

В данном случае после 60 итераций градиентного спуска найден минимум и обучение MLP завершилось. Теперь его можно протестировать на векторах изображений, используя метод `_predict`. Этот метод принимает массив входных данных, то есть массив 64-мерных векторов, и для каждого возвращает выходной вектор. Например, `mlp._predict(x)` выдаст 10-мерные выходные векторы для всех 1000 векторов изображений, хранящихся в x. Результатом для нулевого обучающего примера будет нулевой вектор результата:

```
>>> mlp._predict(x)[0]
array([9.99766643e-01, 8.43331208e-11, 3.47867059e-06, 1.49956270e-07,
       1.88677660e-06, 3.44652605e-05, 6.23829017e-06, 1.09043503e-04,
       1.11195821e-07, 7.79837557e-05])
```

Нужно немного прищуриться, глядя на эти числа. Первое из них равно 0,9998, а все остальные меньше 0,001. Этот результат верно предсказывает, что нулевой обучающий пример — это изображение цифры 0. Пока все хорошо!

Мы можем написать небольшую функцию-обертку, которая использует этот MLP для классификации одного изображения, принимая 64-мерный вектор изображения и возвращая 10-мерный результат. Поскольку MLP в scikit-learn работает с наборами входных векторов и создает массивы результатов, нам просто нужно поместить входной вектор в список перед передачей в `mlp._predict`:

```
def sklearn_trained_classify(v):
    return mlp._predict([v])[0]
```

На данный момент вектор имеет правильную форму, чтобы качество классификации можно было проверить функцией `test_digit_classify`. Посмотрим, какой процент тестовых изображений цифр он идентифицирует правильно:

```
>>> test_digit_classify(sklearn_trained_classify)
1.0
```

Мы добились потрясающей 100%-ной точности! К этому результату можно отнестись с некоторой долей скепсиса — в конце концов, мы тестируем тот же набор данных, который использовался для обучения нейронной сети. Теоретически при наличии 1210 параметров нейронная сеть могла просто запомнить каждый пример из обучающего набора. Но если протестировать изображения, которые нейронная сеть раньше не видела, то можно понять, что это не так: результаты классификации по-прежнему впечатляют — она правильно классифицирует изображения цифр. Немного поэкспериментировав, я обнаружил, что наша сеть показывает точность 96,2 % на следующих 500 изображениях, имеющихся в наборе данных, и вы можете проверить это самостоятельно, выполнив упражнение 16.11.

16.5.4. Упражнения

Упражнение 16.11. Измените функцию `test_digit_classify` так, чтобы она работала с произвольным диапазоном примеров в тестовом наборе, например, выполняла проверку на следующих 500 примерах, следующих за первой 1000 обучающих примеров.

Решение. Я добавил именованный аргумент `start`, чтобы в нем можно было указать, с какого примера следует начать тестирование. Именованный

аргумент `test_count` по-прежнему задает количество примеров для тестирования:

```
def test_digit_classify(classifier,start=0,test_count=1000):
    correct = 0
    end = start + test_count
    for img, target in zip(digits.images[start:end],
                           digits.target[start:end]):
        v = np.matrix.flatten(img) / 15
        output = classifier(v)
        answer = list(output).index(max(output))
        if answer == target:
            correct += 1
    return (correct/test_count)
```

Вычислить конечный индекс в проверяемом наборе данных

Цикл по данным между начальным и конечным индексами

Обученный мною MLP правильно идентифицирует 96,2 % изображений, которые он не видел в процессе обучения:

```
>>> test_digit_classify(sklearn_trained_classify,start=1000,test_count=500)
0.962
```

Упражнение 16.12. Используя функцию потерь, вычисляющую квадрат расстояния, определите величину потерь случайного MLP на первой 1000 обучающих примеров. Какова величина потерь MLP из scikit-learn?

Решение. Во-первых, напишем функцию, дающую идеальный выходной вектор для данной цифры. Например, для цифры 5 она должна вернуть вектор y, состоящий из нулей и единицы в элементе с индексом 5:

```
def y_vec(digit):
    return np.array([1 if i == digit else 0 for i in range(0,10)])
```

Величина потерь на одном тестовом примере — это квадрат расстояния между идеальным результатом и результатом классификатора, то есть сумма квадратов разностей координат:

```
def cost_one(classifier,x,i):
    return sum([(classifier(x)[j] - y_vec(i)[j])**2 for j in range(10)])
```

Общая величина потерь классификатора — это среднее значение потерь на всей 1000 обучающих примеров:

```
def total_cost(classifier):
    return sum([cost_one(classifier,x[j],y[j]) for j in
range(1000)])/1000.
```

Как и ожидалось, случайно инициализированный MLP с точностью всего 10 % имеет гораздо более высокие потери, чем MLP со 100 %-ной точностью, созданный с помощью библиотеки scikit-learn:

```
>>> total_cost(nn.evaluate)
8.995371023185067
>>> total_cost(sklearn_trained_classify)
5.670512721637246e-05
```

Упражнение 16.13. Мини-проект. Извлеките веса и смещения из `MLPClassifier`, используя его свойства `coefs_` и `intercepts_` соответственно. Вставьте эти веса и смещения в класс MLP, который мы построили с нуля в этой главе, и покажите, что получившаяся модель MLP хорошо справляется с классификацией цифр.

Решение. Если вы попробуете решить эту задачу самостоятельно, то заметите одну проблему: мы ожидаем, что весовые матрицы будут иметь размеры 16×64 и 10×16, а свойство `coefs_` экземпляра `MLPClassifier` дает матрицу 64×16 и 16×10. Похоже, что scikit-learn применяет другое соглашение, храня весовые матрицы по столбцам, а не как мы — по строкам. Эту проблему легко исправить.

Массивы NumPy имеют свойство `T`, возвращающее *транспонированную* матрицу (матрицу, полученную поворотом исходной матрицы так, что строки становятся столбцами). Использовав эту хитрость, мы можем включить веса и смещения в нейронную сеть и протестировать ее:

Записать в нашу весовую матрицу значения из весовой матрицы модели scikit-learn после транспонирования для приведения в соответствие с принятыми соглашениями

```
>>> nn = MLP([64,16,10])
>>> nn.weights = [w.T for w in mlp.coefs_]
>>> nn.biases = mlp.intercepts_
>>> test_digit_classify(nn.evaluate,
                        start=1000,
                        test_count=500) 0.962
```

Записать в наш вектор смещений значения из вектора смещений модели scikit-learn

Проверить, насколько хорошо наша нейронная сеть справляется с классификацией после ее инициализации новыми весами и смещениями

Это — точность 96,2 % на 500 изображениях, следующих за набором обучающих данных. Она точно такая же, какую показала модель MLP, созданная с помощью scikit-learn.

16.6. РАСЧЕТ ГРАДИЕНТОВ В ХОДЕ ОБРАТНОГО РАСПРОСТРАНЕНИЯ

Этот раздел читать не обязательно. Вы уже знаете, как обучить MLP с помощью scikit-learn, и готовы решать практические задачи. Вы можете протестировать нейронные сети разных форм и размеров на задачах классификации и поэкспериментировать с их структурой, чтобы повысить эффективность классификации. Поскольку это последний раздел книги, я хотел познакомить вас с последней порцией сложной (но посильной!) математики — вычислением частных производных функции потерь вручную.

Процесс вычисления частных производных в MLP называется *обратным распространением*, потому что начинается с весов и смещений последнего слоя и движется в обратном направлении. Обратное распространение можно разбить на четыре этапа: вычисление производных относительно весов последнего слоя, смещений последнего слоя, весов скрытых слоев и смещений скрытых слоев. Я покажу, как получить частные производные относительно весов в последнем слое, а вы сможете попробовать использовать этот подход, чтобы сделать все остальное.

16.6.1. Вычисление потерь в терминах весов последнего слоя

Обозначим L индекс последнего слоя в MLP. Это означает, что последняя весовая матрица состоит из весов w_{ij}^l, где $l = L$, другими словами, весов w_{ij}^L. Смещения в этом слое будут обозначаться b_j^L, а активации — a_j^L.

Формула вычисления активации a_j^L j-го нейрона в последнем слое — это сумма вклада каждого нейрона в слое $L - l$ с индексом i. Она выглядит так:

$$a_j^L = \sigma\left(b_j^L + \textbf{Сумма}\left[w_{ij}^L a_i^{L-1}\right] \textbf{для каждого значения } i\right).$$

Сумма подсчитывается по всем значениям i от единицы до количества нейронов в слое $L - l$. Обозначим количество нейронов в слое l как n_l, где i изменяется от l до n^{L-1}. На языке математики эта сумма записывается так:

$$\sum_{i=1}^{n_L} w_{ij}^L a_i^{L-1}.$$

Выражаясь простым человеческим языком, эта формула говорит: «для фиксированных значений L и j сложить значения выражений $w_{ij}^L a_i^{L-1}$ для каждого i от единицы до n_L». Это не что иное, как формула умножения матриц, записанная в виде суммы. Вычисление активации с использованием этой формы выглядит так:

$$a_j^L = \sigma\left(b_j^L + \sum_{i=1}^{n_{L-1}} w_{ij}^L a_i^{L-1}\right).$$

Учитывая реальный опыт обучения, полученный ранее, у нас может иметься некоторый идеальный выходной вектор y с единицей в правильном элементе и с нулями в других элементах. Величина потерь — это квадрат расстояния между вектором активации a_j^L и идеальными выходными значениями y_j. То есть

$$C = \sum_{j=1}^{n_L}\left(a_j^L - y_j\right).$$

Веса w_{ij}^L оказывают косвенное влияние на C. Сначала они умножаются на активации из предыдущего слоя, складываются со смещениями, пропускаются через сигмоиду, а затем передаются на вход квадратичной функции потерь. К счастью, в главе 10 мы узнали, как получить производную композиции функций. Рассматриваемый далее пример немного сложнее, но вы наверняка сможете распознать в нем то же цепное правило, которое видели раньше.

16.6.2. Вычисление частных производных для весов последнего слоя с помощью цепного правила

Разобьем переход от w_{ij}^L к C на три шага. Прежде всего вычислим значение для передачи в сигмоиду, которое мы обозначили как z_j^L ранее в этой главе:

$$z_j^L = b_j^L + \sum_{i=1}^{n_{L-1}} w_{ij}^L a_i^{L-1}.$$

Затем передадим z_j^L сигмоидной функции, чтобы получить активацию a_j^L:

$$a_j^L = \sigma\left(z_j^L\right).$$

И наконец, вычислим величину потерь:

$$C = \sum_{j=1}^{n_L}\left(a_j^L - y_j\right)^2.$$

Чтобы найти частную производную C по w_{ij}^L, перемножим производные этих трех выражений, участвующих в «композиции». Производная z_j^L относительно одного веса w_{ij}^L представляет собой произведение этого веса на активацию a_j^{L-1}. Напоминает производную $y(x) = ax$ относительно x, которая является константой a. Частная производная

$$\frac{\partial z_j^L}{\partial w_{ij}^L} = a_i^{L-1}.$$

Следующий шаг — применение сигмоидной функции, поэтому производная от a_j^L относительно z_j^L — это производная от σ. Оказывается (в разделе с упражнениями вам будет предоставлена возможность подтвердить это), что производная от $\sigma(x)$ равна $\sigma(x)(1 - \sigma(x))$. Эта красивая формула отчасти вытекает из того факта, что e^x одновременно является своей собственной производной. Это дает нам

$$\frac{da_j^L}{dz_j^L} = \sigma'\left(z_j^L\right) = \sigma\left(z_j^L\right)\left(1 - \sigma\left(z_j^L\right)\right).$$

Это обычная, а не частная производная, потому что a_j^L — функция только одной переменной, z_j^L. Наконец, нам нужна производная C относительно a_j^L. Только один член суммы зависит от w_{ij}^L, поэтому нужна только производная от $\left(a_j^L - y_j\right)^2$ относительно a_j^L. В этом контексте y_j — константа, поэтому производная равна $2a_j^L$. Это вытекает из степенного правила, согласно которому, если $f(x) = x^2$, то $f'(x) = 2x$. Для последней производной понадобится

$$\frac{\partial C}{\partial a_j^L} = 2\left(a_j^L - y_j\right).$$

Версия цепного правила с несколькими переменными утверждает, что

$$\frac{\partial C}{\partial w_{ij}^L} = \frac{\partial C}{\partial a_j^L}\frac{da_j^L}{dz_j^L}\frac{\partial z_j^L}{w_{ij}^L}.$$

Эта формула выглядит немного иначе, чем версия, которую мы видели в главе 10, где рассматривалась только композиция из двух функций одной переменной. Однако принцип тот же: записав C в терминах a_j^L, a_j^L — в терминах z_j^L и z_j^L — в терминах w_{ij}^L, мы получаем C, записанное в терминах w_{ij}^L. Цепное правило гласит, что для получения производной всей цепочки нужно перемножить производные каждого члена композиции. Подставляя производные, получаем:

$$\frac{\partial C}{\partial w_{ij}^L} = 2\left(a_j^L - y_j\right) \cdot \sigma\left(z_j^L\right)\left(1 - \sigma\left(z_j^L\right)\right) \cdot a_i^{L-1}.$$

Это одна из четырех формул, которые нужны, чтобы найти весь градиент C. В частности, она дает частную производную для любого веса в последнем слое. Всего их 16×10, то есть мы рассмотрели 160 из 1210 частных производных, необходимых для получения полного градиента.

Причина того, что на этом я остановлюсь, заключается в следующем: производные других весов требуют более сложного применения цепного правила. Любая активация влияет на каждую последующую активацию в нейронной сети, поэтому каждый вес влияет на каждую последующую активацию. Не могу сказать, что это выше вашего понимания, но чувствую, что должен был дать более полное объяснение цепного правила с несколькими переменными, прежде погружаться в кровавые подробности. И если мне удалось вас заинтересовать, следите за появлением продолжения этой книги (скрещиваю пальцы). Благодарю за внимание!

16.6.3. Упражнения

Упражнение 16.14. Мини-проект. Используйте SymPy или наш код из главы 10, чтобы автоматически найти производную сигмоидной функции

$$\sigma(x) = \frac{1}{1+e^{-x}}.$$

Покажите, что она равна $\sigma(x)(1 - \sigma(x))$.

Решение. Библиотека SymPy позволяет быстро получить формулу производной:

```
>>> from sym-py import *
>>> X = symbols('x')
>>> diff(1 / (1+exp(-X)),X)
exp(-x)/(1 + exp(-x))**2
```

На языке математики эти вычисления записываются так:

$$\frac{e^{-x}}{\left(1+e^{-x}\right)^2} = \frac{e^{-x}}{1+e^{-x}} \cdot \frac{1}{1+e^{-x}} = \frac{e^{-x}}{1+e^{-x}} \cdot \sigma(x).$$

Чтобы показать, что это выражение равно $\sigma(x)(1 - \sigma(x))$, необходимо выполнить некоторые вычисления и применить алгебраические правила, но я считаю, что вам стоит убедиться в верности этой формулы. Умножая числитель и знаменатель на e^x и учитывая, что $e^x \cdot e^{-x} = 1$, получаем:

$$\frac{e^{-x}}{\left(1+e^{-x}\right)^2} = \frac{1}{e^x+1} \cdot \sigma(x) = \frac{e^{-x}}{e^{-x}} \cdot \frac{1}{e^x+1} \cdot \sigma(x) =$$

$$= \frac{e^{-x}}{1+e^{-x}} \cdot \sigma(x) = \left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right) \cdot \sigma(x) =$$

$$= \left(1 - \frac{1}{1+e^{-x}}\right) \cdot \sigma(x) = \left(1 - \sigma(x)\right) \cdot \sigma(x).$$

КРАТКИЕ ИТОГИ ГЛАВЫ

- Искусственная нейронная сеть — это математическая функция, вычисление которой отражает поток сигналов в человеческом мозгу. Как функция она принимает вектор на входе и возвращает другой вектор на выходе.
- Нейронную сеть можно использовать для классификации векторных данных, например, изображений, преобразованных в векторы значений пикселов.

Выход нейронной сети — это вектор чисел, говорящих о том, что входной вектор можно отнести к каждому из возможных классов.

- Многослойный перцептрон — это особый вид искусственной нейронной сети, состоящей из нескольких упорядоченных слоев нейронов, в которой нейроны каждого слоя связаны с нейронами предыдущего слоя и находятся под их влиянием. В процессе вычислений каждый нейрон получает числовое значение, являющееся его активацией. Активации можно рассматривать как промежуточные ответы «да» или «нет» на пути к решению задачи классификации.
- Нейроны первого слоя нейронной сети получают свои активации из значений элементов входного вектора. В каждом последующем слое активации вычисляются как функция предыдущего слоя. Последний слой обрабатывается как вектор и возвращается как результат вычислений.
- Активация нейрона основана на линейной комбинации активаций всех нейронов предыдущего слоя. Коэффициенты в линейной комбинации называются *весами*. Каждый нейрон имеет также *смещение* — число, которое прибавляется к линейной комбинации. Сумма линейной комбинации весов и смещений передается через сигмоидную функцию, чтобы получить функцию активации.
- Под обучением нейронной сети подразумевается настройка значений всех весов и смещений так, чтобы она максимально эффективно решала свою задачу. Для этого можно измерить ошибку прогнозов нейронной сети относительно фактических ответов из набора обучающих данных с помощью функции потерь. При фиксированном наборе обучающих данных функция потерь зависит только от весов и смещений.
- Градиентный спуск позволяет искать значения весов и смещений, минимизирующих функцию потерь и дающих наиболее эффективную нейронную сеть.
- Эффективное обучение нейронных сетей возможно благодаря существованию простых и точных формул частных производных функции потерь относительно весов и смещений. Их можно найти с помощью алгоритма обратного распространения.
- Библиотека scikit-learn для Python имеет встроенный класс `MLPClassifer`, который может автоматически обучаться на классифицированных векторных данных.

Приложение А Подготовка к работе с Python

В этом приложении описаны основные шаги по установке Python и дополнительных инструментов, что позволит опробовать примеры кода из книги. Прежде всего нужно установить Anaconda — популярный дистрибутив Python для математического программирования и обработки данных. В состав Anaconda входит интерпретатор, который выполняет код на Python, а также ряд популярных математических библиотек, библиотек обработки данных и интерфейс программирования под названием Jupyter. Шаги в основном одинаковы на любом компьютере с Linux, Mac или Windows. Я покажу, какие шаги выполнял на своем Mac.

А.1. ПРОВЕРКА НАЛИЧИЯ PYTHON В СИСТЕМЕ

Возможно, на вашем компьютере уже установлен Python, даже если вы об этом не подозревали. Для проверки откройте окно терминала (командную строку или PowerShell в Windows) и введите команду `python`. На Mac прямо на заводе устанавливается Python 2.7. Чтобы выйти из интерактивного сеанса Python и закрыть терминал, нажмите комбинацию Ctrl+D.

```
paul — python — 80×24
Pauls-MacBook-Pro:~ paul$ python
Python 2.7.10 (default, Feb 22 2019, 21:55:15)
[GCC 4.2.1 Compatible Apple LLVM 10.0.1 (clang-1001.0.37.14)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Для разработки примеров в этой книге я использовал Python 3, который постепенно становится новым стандартом, в частности, дистрибутив Anaconda. Предупреждаю: если в вашей системе уже установлен Python, то следующие шаги могут вызвать некоторые сложности. Если какая-либо из приведенных далее инструкций у вас не сработает, рекомендую выполнить поиск по тексту сообщения об ошибке в Google или StackOverflow.

Если вы имеете богатый опыт работы в Python и не хотите устанавливать или использовать Anaconda, то найдите и установите необходимые библиотеки, такие как NumPy, Matplotlib и Jupyter, с помощью диспетчера пакетов `pip`. Начинающим же я настоятельно рекомендую установить Anaconda, как описывается далее.

А.2. ЗАГРУЗКА И УСТАНОВКА ANACONDA

Откройте в браузере страницу https://www.anaconda.com/distribution/, щелкните на кнопке Download (Загрузить) и выберите версию Python, начинающуюся с 3 (рис. А.1). На момент написания книги самой свежей была версия Python 3.7.

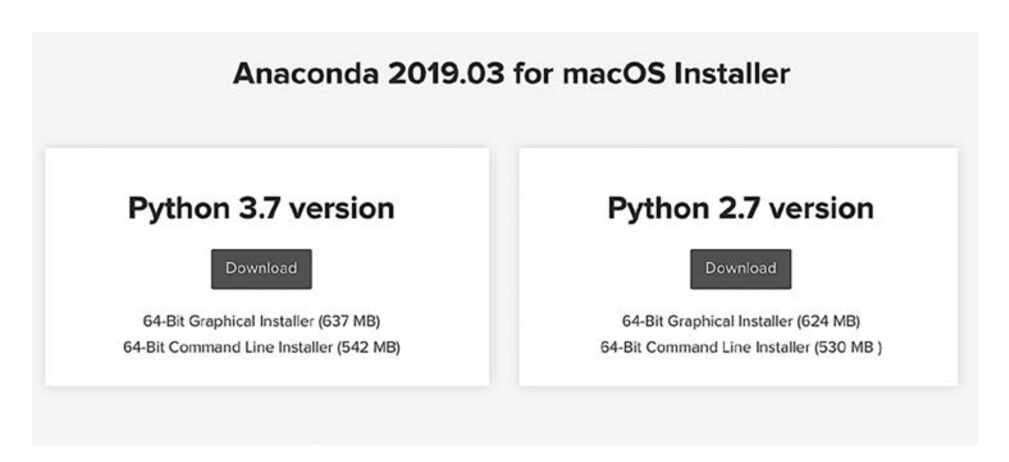


Рис. А.1. Когда я работал над книгой, после нажатия кнопки Download (Загрузить) отображалась эта страница. Для установки Python выберите ссылку download под заголовком Python 3.x

После загрузки запустите программу установки. Она проведет вас через процесс установки. Диалоговое окно мастера установщика выглядит по-разному в разных операционных системах, на рис. А.2 показано, какое оно на Mac.



Рис. А.2. Окно мастера установки Anaconda на Mac

Я оставил выбор места установки по умолчанию и не добавлял установку никаких дополнительных продуктов, таких как PyCharm IDE. После завершения установки откройте новое окно терминала и введите команду `python`, чтобы запустить интерактивный сеанс Python 3 с Anaconda (рис. А.3).

```
paul — python — 80×24
(base) Pauls-MacBook-Pro:~ paul$ python
Python 3.7.3 (default, Mar 27 2019, 16:54:48)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Рис. А.3. Так должен выглядеть интерактивный сеанс Python после установки Anaconda. Обратите внимание на появившиеся метки Python 3.7.3 и Anaconda, Inc.

Если вы не увидели версии Python, начинающейся с цифры 3, и слова Anaconda, это может означать, что предустановленная версия Python в вашей системе обнаруживается командной оболочкой раньше, чем новая. Отредактируйте переменную окружения `PATH`, чтобы подсказать командной оболочке, какую версию

Python нужно запускать при вводе команды python. Надеюсь, вы не столкнетесь с этой проблемой, но если такое случится, поищите решение в Интернете. Вместо команды python можно попробовать ввести команду python3, чтобы явно использовать вновь установленную версию Python 3.

А.3. ПРИМЕНЕНИЕ PYTHON В ИНТЕРАКТИВНОМ РЕЖИМЕ

Три угловые скобки (>>>) в окне терминала предлагают ввести строку кода на Python. Набрав 2+2 и нажав Enter, вы должны увидеть результат вычисления этого выражения интерпретатором Python, который равен 4 (рис. А.4).

```
paul — python — 80×24
(base) Pauls-MacBook-Pro:~ paul$ python
Python 3.7.3 (default, Mar 27 2019, 16:54:48)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2
4
>>>
```

Рис. А.4. Ввод строки кода на Python в интерактивном сеансе

Интерактивный режим называется также циклом REPL (read — evaluate — print — loop — «прочитать — вычислить — напечатать — повторить»). Интерактивный сеанс Python читает введенную строку кода, вычисляет ее и выводит результат. Этот процесс может повторяться до бесконечности. Нажав комбинацию Ctrl+D, вы сообщите, что закончили ввод кода, после чего интерактивный сеанс Python завершится и вы вернетесь в сеанс терминала.

Интерактивный сеанс Python обычно без затруднений определяет ввод многострочных операторов. Например, def f(x): — это первая строка, которую вы вводите, чтобы определить новую функцию f. Интерактивный сеанс Python покажет ..., чтобы сообщить, что ждет ввода дополнительного кода (рис. А.5).

```
>>> def f(x):
...
```

Рис. А.5. Интерпретатор Python знает, что вы еще не завершили ввод многострочного оператора

Можете добавить отступ, чтобы продолжить определение функции, а по окончании дважды нажать Enter, чтобы сообщить интерпретатору, что ввод многострочного кода завершен (рис. А.6).

```
[>>> def f(x):
[...     return x * x
[...
>>>
```

Рис. А.6. Закончив ввод многострочного оператора, дважды нажмите Enter, чтобы передать его для выполнения интерпретатору

Теперь функция `f` определена в интерактивном сеансе. В следующей строке можете попробовать вызвать ее (рис. А.7).

```
[>>> f(5)
25
```

Рис. А.7. Вызов ранее определенной функции

Имейте в виду, что любой код, который вы введете в интерактивном сеансе, исчезнет сразу после выхода из него. Поэтому, если собираетесь написать много кода, лучше поместить его в файл сценария или в блокнот Jupyter. Далее я опишу оба этих метода.

А.3.1. Создание и запуск файла сценария на Python

Создать сценарий на Python можно практически в любом текстовом редакторе. Как правило, стоит использовать текстовые редакторы, предназначенные для программирования, а не текстовые процессоры, такие как Microsoft Word, которые могут вставлять невидимые или нежелательные символы для форматирования. Я предпочитаю Visual Studio Code. В числе других популярных вариантов можно назвать кросс-платформенный Atom и Notepad++ для Windows. Можно использовать и текстовые редакторы, действующие в терминале, такие как Emacs или Vim. Все эти инструменты бесплатны и легко доступны.

Чтобы написать сценарий на Python, создайте в редакторе новый текстовый файл с расширением `.py`. На рис. А.8 показано, что я создал файл `first.py` в каталоге `~/Documents`. На рис. А.8 также видно, что текстовый редактор Visual Studio Code поддерживает подсветку синтаксиса для Python. Ключевые слова, функции и литеральные значения окрашены в определенные цвета, чтобы облегчить чтение кода. Многие редакторы, включая Visual Studio Code, имеют дополнительные расширения, которые можно установить, чтобы получить в свое распоряжение полезные инструменты, например, проверяющие появление простых ошибок при вводе текста.

```
first.py
Users ▸ paul ▸ Documents ▸ first.py ▸ ...
def f(x):
    return x * x

for x in range(0,10):
    print(f(x))
```

Рис. А.8. Пример кода на Python в файле. Этот код выводит квадраты всех чисел от 0 до 9

На рис. А.8 показано несколько строк кода на Python в файле `first.py`. Поскольку это книга посвящена математике, мы можем использовать более «математический» пример, чем Hello World. После запуска этот код выводит квадраты всех чисел от 0 до 9.

Закончив ввод кода, сохранив файл и вернувшись в терминал, перейдите в каталог с этим файлом. На Mac я перешел в каталог `~/Documents`, введя команду `cd ~/Documents`. После этого можно ввести команду `ls first.py`, чтобы убедиться, что вы действительно перешли в каталог с файлом сценария на Python (рис. А.9).



Рис. А.9. Команда ls показывает, что в текущем каталоге присутствует файл first.py

Чтобы выполнить сценарий, введите команду `python first.py` в окне терминала. Она вызовет интерпретатор Python и сообщит ему имя файла `first.py`, который нужно выполнить. Интерпретатор сделает ровно то, на что мы рассчитывали, и выведет последовательность чисел (рис. А.10).

```
(base) Pauls-MacBook-Pro:Documents paul$ python first.py
0
1
4
9
16
25
36
49
64
81
```

Рис. А.10. Результат запуска простого сценария на Python из командной строки

При решении более сложных задач может понадобиться разбить код на отдельные файлы. Далее я покажу, как поместить функцию `f(x)` в другой файл с кодом на Python, который затем можно использовать из `first.py`. Создадим новый файл `function.py` и сохраним его в том же каталоге, где находится `first.py`, затем скопируем в него код, определяющий функцию `f(x)` (рис. А.11), и удалим этот код из `first.py`.

```
def f(x):
    return x * x
```

Рис. А.11. Размещение кода с определением функции *f*(*x*) в отдельном файле

Чтобы сообщить интерпретатору Python, что вы собираетесь объединить несколько файлов в этом каталоге, добавьте в каталог пустой текстовый файл `__init__.py`. (Здесь по два символа подчеркивания до и после слова `init`.)

СОВЕТ

Создать пустой файл на компьютере Mac или Linux можно командой `touch __init__.py`.

Чтобы использовать функцию `f(x)` из `function.py` в сценарии `first.py`, нужно сообщить интерпретатору Python, как ее получить. Для этого добавим в начало файла `first.py` строку `from function import f` (рис. А.12).



Рис. А.12. Добавление в начало файла first.py новой инструкции для включения функции f(x)

Теперь, выполнив команду `python first.py`, вы должны получить тот же результат, что и в прошлый раз. Но на этот раз Python получит функцию `f` из `function.py`.

Альтернатива размещению кода в текстовых файлах и запуску их из командной строки — использование блокнотов Jupyter, о которых я расскажу далее. Бóльшую часть примеров для книги я создал в блокнотах Jupyter, но весь повторно применяемый код поместил в отдельные файлы на Python и импортировал их.

А.3.2. Использование блокнотов Jupyter

Jupyter Notebook — это графический интерфейс для программирования на Python и других языках. Так же как в интерактивном сеансе Python, вы вводите строки кода в блокнот Jupyter, а он выводит результат. Разница лишь в том, что интерфейс Jupyter красивее аскетичного терминала и позволяет сохранять сеансы, чтобы вернуться к ним позже.

Jupyter Notebook должен автоматически устанавливаться при установке из дистрибутива Anaconda. Если вы используете другой дистрибутив Python, то для установки Jupyter можете применить `pip`. На странице с документацией https://jupyter.org/install вы найдете дополнительные инструкции, если решите выполнить выборочную установку.

Чтобы открыть интерфейс Jupyter Notebook, введите команду `jupyter notebook` или `python -m notebook` в терминале, предварительно перейдя в рабочий каталог. Вы должны увидеть в терминале нескончаемый поток текста, а ваш веб-браузер по умолчанию должен открыть интерфейс Jupyter Notebook.

На рис. А.13 показано, что отображается в моем терминале после ввода команды `python -m notebook`. И снова то, что увидите вы, может отличаться в зависимости от вашей версии Anaconda.

```
Documents — python -m notebook — 96×24
(base) Pauls-MacBook-Pro:Documents paul$ python -m notebook
[I 17:43:05.088 NotebookApp] Writing notebook server cookie secret to /Users/paul/Library/Jupyte
r/runtime/notebook_cookie_secret
[I 17:43:06.476 NotebookApp] JupyterLab extension loaded from /anaconda3/lib/python3.7/site-pack
ages/jupyterlab
[I 17:43:06.476 NotebookApp] JupyterLab application directory is /anaconda3/share/jupyter/lab
[I 17:43:06.478 NotebookApp] Serving notebooks from local directory: /Users/paul/Documents
[I 17:43:06.479 NotebookApp] The Jupyter Notebook is running at:
[I 17:43:06.479 NotebookApp] http://localhost:8888/?token=1b9b9027842917638b940628263cd4ebeb9577
e5b0878a28
[I 17:43:06.479 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice
to skip confirmation).
[C 17:43:06.486 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///Users/paul/Library/Jupyter/runtime/nbserver-2056-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=1b9b9027842917638b940628263cd4ebeb9577e5b0878a28
```

Рис. А.13. Так выглядит окно терминала, когда открывается блокнот Jupyter

Ваш веб-браузер по умолчанию должен открыть интерфейс Jupyter. На рис. А.14 показано, как он выглядит у меня в браузере Google Chrome.

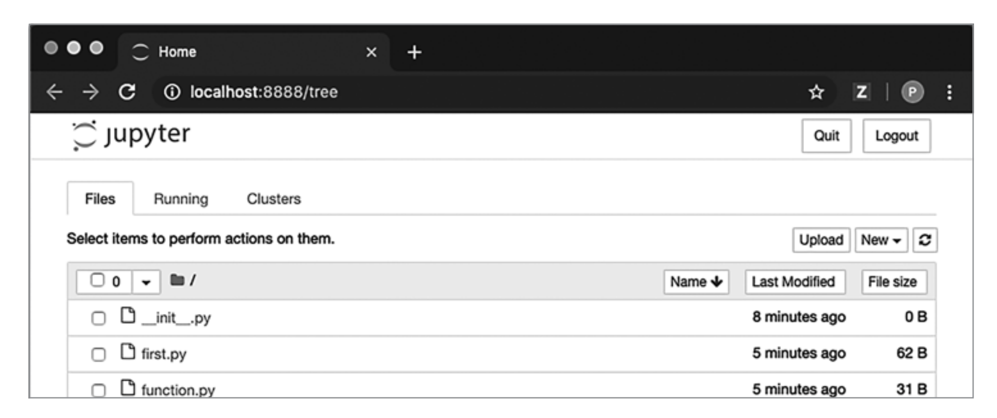


Рис. А.14. После запуска Jupyter автоматически откроется вкладка браузера, которая выглядит примерно так

За кулисами терминал продолжает работать под управлением Python, а также обслуживает локальный веб-сайт, доступный по адресу localhost:8888. С этого момента думайте только о том, что происходит в браузере. Браузер автоматически отправляет написанный вами код процессу Python, выполняющемуся в терминале, используя веб-запросы. В терминологии Jupyter этот фоновый процесс Python называется *ядром*.

На первой вкладке, которая открывается в браузере, можно видеть все файлы, содержащиеся в рабочем каталоге. Например, я открыл блокнот в папке `~/Documents`, где находятся файлы на Python, которые мы написали в предыдущем разделе. Если щелкнуть на одном из файлов, он откроется и станет доступен для просмотра и редактирования прямо в веб-браузере. На рис. А.15 показано, как выглядит окно браузера после щелчка на `first.py`.

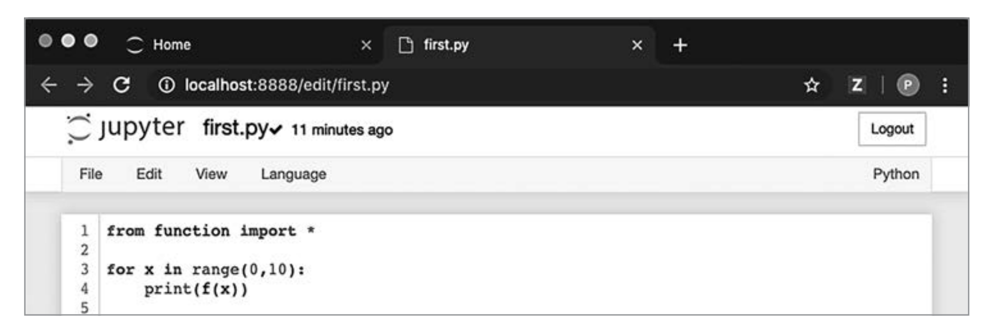


Рис. А.15. Jupyter поддерживает простой текстовый редактор для файлов на Python. В данном случае я открыл файл first.py

Это еще не блокнот. Блокнот — это файл другого типа, отличный от обычного файла с кодом на Python. Чтобы создать блокнот, вернитесь на главную страницу, щелкнув на логотипе Jupyter в верхнем левом углу, затем в раскрывающемся меню New (Создать) справа выберите пункт Python 3 (рис. А.16).

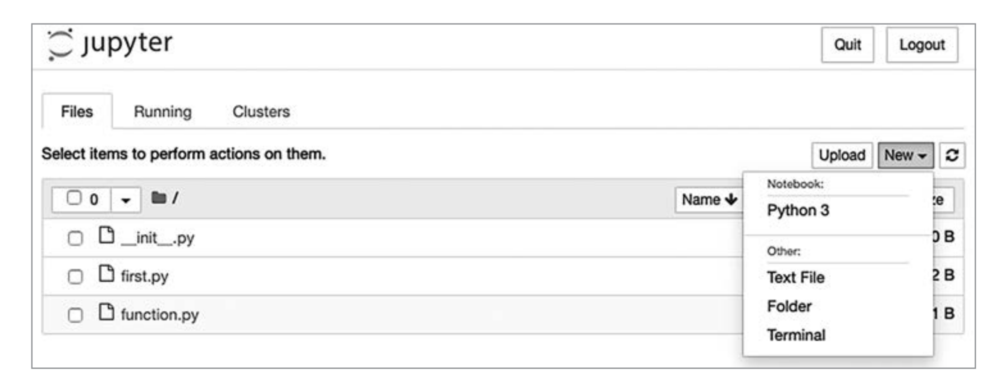


Рис. А.16. Выбор пункта меню для создания нового блокнота Python 3

После выбора пункта Python 3 откроется новый блокнот. Он должен выглядеть так, как показано на рис. А.17: с одной пустой строкой ввода, готовой принять код на Python.

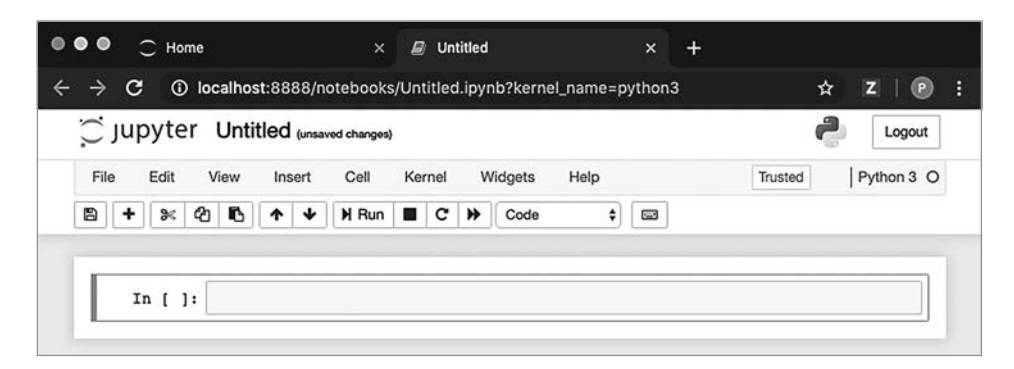


Рис. А.17. Новый пустой блокнот Jupyter, готовый к программированию

Вы можете ввести в текстовое поле выражение на Python, а затем нажать Shift+Enter, чтобы вычислить его. На рис. А.18 показано, как я набрал 2+2, а затем нажал Shift+Enter, чтобы увидеть результат 4.

Как видите, блокнот работает точно так же, как интерактивный сеанс, только выглядит симпатичнее. Каждый введенный код отображается в рамке, а соответствующий вывод находится под ним.

```
In [1]: 2 + 2
Out[1]: 4

In [ ]:
```

Рис. А.18. Вычисление выражения 2 + 2 в блокноте Jupyter

Если просто нажать Enter вместо Shift+Enter, то произойдет переход на новую строку в поле ввода. Переменные и функции, определенные в полях, расположенных выше, могут использоваться в полях, находящихся ниже. На рис. А.19 показано, как наш первоначальный пример мог бы выглядеть в блокноте Jupyter.

```
In [1]: 2 + 2
Out[1]: 4

In [2]: def f(x):
            return x * x

In [3]: for i in range(0,10):
            print(f(i))

        0
        1
        4
        9
        16
        25
        36
        49
        64
        81

In [ ]:
```

Рис. А.19. Ввод и выполнение нескольких фрагментов кода на Python в блокноте Jupyter. Обратите внимание на поля ввода и результаты

Строго говоря, каждый блок зависит не от блоков над ним, а от блоков, которые были *выполнены последними*. Например, если я переопределю функцию `f(x)` в следующем поле ввода, а затем повторно выполню предыдущее, то предыдущий вывод будет перезаписан (рис. А.20).

Такое поведение может сбивать с толку, поэтому Jupyter пытается помочь, перенумеровывая поля ввода по мере их выполнения. Для большей надежности я предлагаю определять переменные и функции перед их первым использованием. Вы можете убедиться, что код работает правильно сверху вниз, выбрав пункт меню Kernel ▸ Restart & Run All (Ядро ▸ Перезапустить и запустить все) (рис. А.21).

В этом случае Jupyter сотрет все полученные прежде результаты, но если вы будете последовательны, то получите то же самое.

```
In [1]: 2 + 2
Out[1]: 4

In [2]: def f(x):
            return x * x

In [5]: for i in range(0,10):
            print(f(i))

        0
        1
        8
        27
        64
        125
        216
        343
        512
        729

In [4]: def f(x):
            return x * x * x

In [ ]:
```

Рис. А.20. Если переопределите символ, например f, ниже предыдущего вывода, а затем повторно запустите поле выше, то Python использует новое определение функции. Сравните этот рисунок с рис. А.19, чтобы увидеть новую ячейку

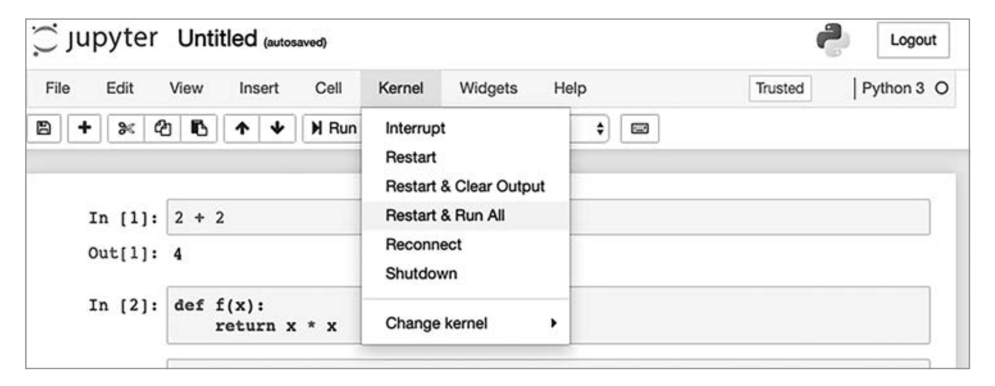


Рис. А.21. Используйте пункт меню Kernel ▸ Restart & Run All (Ядро ▸ Перезапустить и запустить все), чтобы очистить прежние результаты и выполнить весь введенный код заново сверху вниз

Блокнот будет сохраняться автоматически. Закончив работу, можете дать имя блокноту, щелкнув на ссылке Untitled (Без названия) в верхней части экрана и введя новое имя (рис. А.22).

Рис. А.22. Присвоение имени блокноту

Затем можно еще раз щелкнуть на логотипе Jupyter, чтобы вернуться в главное меню, после чего вы сможете увидеть новый блокнот, сохраненный в виде файла с расширением `.ipynb` (рис. А.23). Чтобы вновь открыть блокнот, просто щелкните на его имени.

Рис. А.23. В списке файлов появился новый блокнот Jupyter

СОВЕТ

> Чтобы гарантировать сохранность файлов, выходите из Jupyter щелчком на кнопке Quit (Выход), а не просто закрыв вкладку браузера или остановив интерактивный процесс в терминале.

За дополнительной информацией о блокнотах Jupyter обращайтесь к обширнейшей документации, доступной по адресу https://jupyter.org/. Однако вы уже знаете достаточно, чтобы загрузить примеры исходного кода для этой книги, организованные в виде блокнотов Jupyter почти для всех глав, и поэкспериментировать с ними.

Приложение Б Советы и рекомендации по работе с Python

Следуя инструкциям по подготовке, приведенным в приложении А, вы сможете установить и настроить Python на своем компьютере, подготовившись к экспериментам с программным кодом. Если вы только начинаете осваивать Python, то следующим вашим шагом должно стать знакомство с некоторыми особенностями языка. Даже если вам не приходилось программировать на Python раньше, не переживайте! Это один из самых простых и доступных для изучения языков программирования. Кроме того, существует множество отличных онлайн-ресурсов и книг, которые помогут вам изучить основы программирования на Python, и прекрасной отправной точкой послужит веб-сайт python.org.

В этом приложении я предполагаю, что у вас уже есть некоторый опыт работы с Python и вы знакомы с его основами, такими как числа, строки, `True` и `False`, операторы `if`/`else` и т. д. Чтобы сделать эту книгу как можно более доступной, я старался избегать использования расширенных возможностей Python. Но в этом приложении познакомлю вас с некоторыми возможностями этого языка, которые либо выходят за рамки основ, либо заслуживают особого внимания из-за их важности для книги. Не волнуйтесь, если что-то покажется вам сложным: когда эти возможности появляются в книге, я обычно добавляю краткий обзор особенностей их работы. Весь код в этом приложении описан в блокноте walkthrough в примерах исходного кода.

Б.1. ЧИСЛА И МАТЕМАТИКА В PYTHON

Как и большинство языков программирования, Python имеет встроенную поддержку основных математических действий. Я полагаю, что вы уже знакомы с его основными арифметическими операторами: `+`, `-`, `*` и `/`. Обратите внимание

на то, что при делении целых чисел в Python 3 может получиться дробное значение, например,

```
>>> 7/2
3.5
```

В Python 2, напротив, эта операция вернула бы `2` — результат целочисленного деления с отбрасыванием остатка 1. Но иногда бывает нужно получить остаток, и в таких случаях можно использовать оператор `%`, называемый оператором *модуля* (деления по модулю). Выражение `13 % 5` даст в результате 3, сообщая, что целочисленное деление 13 на 5 дает в остатке 3 (например, $13 = 2 \cdot 5 + 3$). Обратите также внимание на то, что оператор модуля может работать с числами с плавающей точкой. В частности, с его помощью можно получить дробную часть числа как остаток от деления на 1. Выражение `3.75 % 1` даст в результате `0.75`.

Еще один полезный математический оператор — `**`, который выполняет возведение в степень. Например, `2 ** 3` — это два в третьей степени, или 2^3, что равно 8. Аналогично `4 ** 2` — это 4^2, что равно 16.

И последнее, о чем следует помнить, занимаясь математикой в Python, — арифметика с плавающей точкой имеет ограниченную точность. Я не буду вдаваться в описание причин, но покажу последствия, чтобы это не стало для вас неприятным сюрпризом. Например, выражение 1000,1 – 1000,0, очевидно, должно давать в результате 0,1, но Python вычисляет это значение неточно:

```
>>> 1000.1 - 1000.0
0.10000000000002274
```

Конечно, этот результат отличается от истинного менее чем на одну триллионную, поэтому он не вызовет у нас проблем, но иногда результаты могут выглядеть неверными. Например, мы ожидаем, что выражение (1000,1 – 1000,0) – 0,1 даст в результате ноль, но вместо этого Python возвращает результат, кажущийся большим:

```
>>> (1000.1 - 1000.0) - 0.1
2.273181642920008e-14
```

Это длинное число записано в экспоненциальной нотации и примерно в 2,27 раза больше 10^{-14}. Число 10^{-14} равно 1/100 000 000 000 000 (1, деленная на 1 с 14 нулями, или 1 на 100 трлн), то есть это число очень близко к нулю.

Б.1.1. Модуль math

В стандартной библиотеке Python имеется модуль `math` с набором полезных математических значений и функций. По аналогии с любым другим модулем Python вы должны импортировать из него объекты, которые предполагаете использовать. Например,

```
from math import pi
```

импортирует переменную `pi` из модуля `math`, которая представляет число π. Возможно, вы помните из геометрии, что π — это отношение длины окружности к ее диаметру. Импортировав значение `pi`, его можно использовать как любую другую переменную:

```
>>> pi
3.141592653589793
>>> tau = 2 * pi
>>> tau
6.283185307179586
```

Другой способ получить доступ к значениям в модулях Python — импортировать сам модуль, а затем применять его для обращения к его значениям. В следующем примере я импортирую модуль `math`, а затем использую его для доступа к числу π и другому специальному числу *e*, с которым не раз столкнемся в этой книге:

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
```

Модуль `math` содержит также ряд важных функций, с которыми мы будем работать в книге. Среди них функция квадратного корня `sqrt`, тригонометрические функции `cos` и `sin`, экспоненциальная функция `exp` и функция натурального логарифма `log`. Мы рассмотрим каждую из этих функций в свое время, а пока достаточно запомнить, что они вызываются как обычные функции Python с передачей им входных значений в круглых скобках:

```
>>> math.sqrt(25)
5.0
>>> math.sin(pi/2)
1.0
>>> math.cos(pi/3)
0.5000000000000001
>>> math.exp(2)
7.38905609893065
>>> math.log(math.exp(2))
2.0
```

В качестве краткого напоминания об экспоненциальных функциях: `math.exp(x)` дает такой же результат, как и выражение `math.e ** x`, для любого значения `x`, а функция `math.log` обращает эффект `math.exp`. Тригонометрические функции представлены в главе 2.

Б.1.2. Случайные числа

Иногда бывает нужно выбрать несколько произвольных чисел для проверки вычислений. Для этого можно использовать генераторы случайных чисел Python. Они находятся в модуле `random`, поэтому сначала его нужно импортировать:

```
import random
```

Первая важная функция в этом модуле — `randint`. Она возвращает случайное целое значение из заданного диапазона. Например, вызов `random.randint(0,10)` вернет случайно выбранное целое число от 0 до 10, при этом обе границы, 0 и 10, являются возможными результатами:

```
>>> random.randint(0,10)
7
>>> random.randint(0,10)
1
```

Еще одна функция, которая используется для генерации случайных чисел, — `random.uniform`. Она генерирует случайное число с плавающей точкой в заданном интервале. Следующий код возвращает случайное число от 7,5 до 9,5:

```
>>> random.uniform(7.5, 9.5)
8.200084576283352
```

Слово *uniform* (равномерный) сообщает, что ни один из поддиапазонов не является более вероятным, чем другие. Напротив, если выбрать случайных людей и узнать их возраст, то получится неравномерное распределение случайных чисел, а это означает, что вы найдете больше людей в возрасте от 10 до 20 лет, чем от 100 до 110 лет.

Б.2. НАБОРЫ ДАННЫХ В PYTHON

На протяжении всей книги мы выполняем математические операции с *наборами* данных. Это могут быть упорядоченные пары чисел, представляющие точки на плоскости, списки чисел, отражающие результаты измерений в реальном мире, или наборы символов в алгебраическом выражении. Python поддерживает несколько способов моделирования наборов, и в этом разделе я перечислю их и представлю сравнительные характеристики.

Б.2.1. Списки

Самый простой набор в Python — это список. Чтобы создать его, нужно просто заключить несколько значений в квадратные скобки, разделив их запятыми. Вот список из трех строк, который хранится в переменной `month`:

```
months = ["January", "February", "March"]
```

Элементы списка доступны по их *индексам* — номерам позиций в списке. Нумерация элементов списков в Python начинается с нуля, а не с единицы. В списке `months` доступны три индекса: 0, 1 и 2. Следовательно, мы можем получить его элементы, как показано далее:

```
>>> months[0]
'January'
>>> months[1]
```

```
'February'
>>> months[2]
'March'
```

При попытке обратиться к элементу списка за пределами диапазона допустимых индексов будет сгенерирована ошибка. Например, попробуйте получить элемент `months[3]` или `months[17]`. Чтобы гарантировать использование допустимых индексов, кое-где в книге я применяю трюк с оператором деления по модулю. Для любого целого числа `n` выражение `month[n % 3]` гарантированно будет правильным, потому что `n % 3` всегда возвращает `0`, `1` или `2`.

Другой способ получить доступ к элементам списка — *распаковать* их. Если вы уверены, что в списке `months` всего три элемента, то их можно извлечь так:

```
j, f, m = months
```

Эта инструкция запишет в переменные `j`, `f` и `m` элементы списка `months`, выбирая их по порядку. В результате получится:

```
>>> j
'January'
>>> f
'February'
>>> m
'March'
```

Еще одна базовая операция, поддерживаемая списками, — *конкатенация* (объединение). Она создает список большего размера. Операция конкатенации выполняется с помощью оператора `+`. Объединение списков `[1, 2, 3]` и `[4, 5, 6]` даст новый список, состоящий из элементов первого списка, за которыми следуют элементы второго:

```
>>> [1,2,3] + [4,5,6]
[1, 2, 3, 4, 5, 6]
```

Еще об индексах и срезах списков

Python позволяет извлекать из списка *срез* (slice) — список со всеми значениями между двумя индексами. Например,

```
>>> months[1:3]
['February', 'March']
```

дает срез, начинающийся с индекса 1 и заканчивающийся индексом 3 (но не включая его) в исходном списке. Еще более нагляден пример, элементы которого равны соответствующим индексам:

```
>>> nums = [0,1,2,3,4,5,6,7,8,9,10]
>>> nums[2:5]
[2, 3, 4]
```

Длину списка можно получить с помощью функции `len`:

```
>>> len(months)
3
>>> len(nums)
11
```

Поскольку элементы списка индексируются, начиная с нуля, последний элемент списка имеет индекс, на единицу меньший длины списка. Вот как можно получить последний элемент списка (например, `nums`):

```
>>> nums[len(nums)-1]
10
```

Для доступа к последнему элементу можно использовать и такую форму записи:

```
>>> nums[-1]
10
```

Аналогично, `nums[-2]` вернет предпоследний элемент списка `nums` — число `9`. Существует много способов применения положительных и отрицательных индексов и срезов. Например, `nums[1:]` вернет все элементы списка, кроме первого (с нулевым индексом), а `nums[3:-1]` вернет элементы `nums`, начиная с индекса 3 и заканчивая предпоследним элементом:

```
>>> nums[1:]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> nums[3:-1]
[3, 4, 5, 6, 7, 8, 9]
```

Не путайте синтаксис среза, включающий два индекса, с извлечением элемента из списка списков, где также используются два индекса. Например, в списке

```
list_of_lists = [[1,2,3],[4,5,6],[7,8,9]]
```

значение 8 находится в третьем (индекс 2) подсписке во втором (индекс 1) элементе, соответственно, выражение `list_of_lists[2][1]` даст в результате `8`.

Итерации по спискам

Часто при выполнении вычислений со списками требуется использовать каждое значение в списке. Это означает *перебор* всех его элементов в цикле. Самый простой способ сделать это — взять *цикл* `for`. Следующий цикл `for` выводит каждое значение из списка `months`:

```
>>> for x in months:
>>>     print('Month: ' + x)
Month: January
Month: February
Month: March
```

Можно также построить новый список, начав с пустого списка и последовательно добавляя в него записи с помощью метода `append`. Следующий код создает пустой

список `squares`, а затем перебирает список `nums`, добавляя квадрат каждого числа из `nums` в конец списка `squares`, вызывая `squares.append`:

```
squares = []
for n in nums:
    squares.append(n * n)
```

По окончании цикла `for` список `squares` будет содержать квадраты всех чисел из `nums`:

```
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Генераторы списков

Python поддерживает специальный синтаксис для итеративного построения списков — *генераторы списков* (list comprehension). Генератор списка — это, по сути, особый вид цикла `for`, заключенный в квадратные скобки и указывающий, что на каждом шаге итерации в список добавляется новый элемент. Синтаксис генераторов списков читается как обычные фразы на английском языке, что упрощает их понимание. Например, следующий генератор списков создает список, состоящий из квадратов чисел, выполняя выражение `x * x` для каждого значения `x` в списке `nums`:

```
>>> [x * x for x in nums]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Внутри генератора можно выполнить итерации по нескольким спискам. Например, следующий код перебирает все возможные сочетания значений из списков `years` и `months`, превращая каждую комбинацию года и месяца в строку:

```
>>> years = [2018,2019,2020]
>>> [m + " " + str(y) for y in years for m in months]
['January 2018',
'February 2018',
'March 2018',
'January 2019',
'February 2019',
'March 2019',
'January 2020',
'February 2020',
'March 2020']
```

Аналогично можно построить список списков, поместив один генератор списков в другой. Добавив еще одну пару квадратных скобок, мы меняем генератор списков так, что он возвращает список для каждого значения в списке `months`:

```
>>> [[m + " " + str(y) for y in years] for m in months]
[['January 2018', 'January 2019', 'January 2020'],
['February 2018', 'February 2019', 'February 2020'],
['March 2018', 'March 2019', 'March 2020']]
```

Б.2.2. Другие итерируемые объекты

В Python, и особенно в Python 3.x, есть несколько других типов наборов. Некоторые из них называются *итерируемыми*, потому что позволяют перебирать их содержимое, как если бы они были списками. Вероятно, наиболее часто в этой книге используются *диапазоны*, например, для построения упорядоченных последовательностей чисел. Так, `range(5,10)` дает последовательность целых чисел, начиная с 5 и заканчивая 10 (но не включая это число). Если попытаться вывести значение диапазона `range(5,10)` само по себе, то вы получите неинтересный результат:

```
>>> range(5,10)
range(5, 10)
```

Однако, несмотря на то что диапазон не отображает входящие в него числа, мы можем выполнить итерации по ним как по элементам списка:

```
>>> for i in range(5,10):
>>>     print(i)
5
6
7
8
9
```

Тот факт, что диапазоны не являются списками, позволяет использовать очень большие диапазоны и не перебирать их целиком от начала до конца. Например, `range(0,1000000000)` определяет диапазон из миллиарда чисел, которые можно перебрать в цикле, но в действительности он не хранит этот миллиард чисел. Он хранит только инструкции для получения чисел во время итераций. Чтобы превратить итерируемый объект, например диапазон, в список, достаточно преобразовать его с помощью функции `list`:

```
>>> list(range(0,10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Списки последовательных целых чисел широко используются на практике, поэтому мы часто применяем функцию `range`. Еще одно замечание об этой функции: некоторые ее аргументы необязательны. Если вызвать ее только с одним аргументом, она автоматически начнет счет с нуля и продолжит возвращать последовательные целые числа, пока не достигнет значения единственного аргумента, а если передать ей три аргумента, то она будет увеличивать каждое следующее значение на это число. Например, `range(10)` вернет значения от 0 до 9, а `range(0,10,3)` — от 0 до 9 с шагом 3:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(0,10,3))
[0, 3, 6, 9]
```

Другой пример функции, возвращающей специальный тип итерируемого объекта, — это функция zip. Она принимает два итерируемых объекта одинаковой длины и возвращает итерируемый объект, состоящий из пар соответствующих элементов входных объектов:

```
>>> z = zip([1,2,3],["a","b","c"])
>>> z
<zip at 0x15fa8104bc8>
>>> list(z)
[(1, 'a'), (2, 'b'), (3, 'c')]
```

Обратите внимание на то, что не все итерируемые объекты поддерживают индексацию: попытка получить элемент z[2] завершится ошибкой, поэтому данный объект сначала нужно преобразовать в список (например, list(z)), чтобы получить третий элемент (list(z)[2]). (Диапазоны поддерживают индексацию, то есть range(5,10)[3] вернет число 8.) Будьте осторожны: после обхода всех элементов в объекте, возвращаемом функцией zip, он перестанет существовать! Поэтому, если вы планируете использовать результат вызова zip повторно, я бы посоветовал сразу же преобразовать его в список.

Б.2.3. Функции-генераторы

Функции-генераторы в языке Python дают возможность создавать итерируемые объекты, которые хранят не все свои значения, а только инструкции для их создания. Это позволяет определять большие или даже бесконечные последовательности значений, не расходуя память. Функцию-генератор можно создать несколькими способами. Самый простой из них — определить обычную функцию, включающую инструкцию yield вместо return. Отличие функции-генератора от простой функции в том, что первая может вернуть много результатов, тогда как обычная функция — только один.

Вот функция-генератор, представляющая бесконечную последовательность целых чисел 0, 1, 2, 3 и т. д. Цикл while продолжается вечно, и в каждой итерации функция возвращает значение переменной x, а затем увеличивает ее на 1.

```
def count():
    x = 0
    while True:
        yield x
        x += 1
```

Несмотря на то что эта функция возвращает бесконечную последовательность чисел, вы можете вызвать count(), не опасаясь переполнить память своего компьютера, потому что она возвращает объект генератора, а не полный список значений:

```
>>> count()
<generator object count at 0x0000015FA80EC750>
```

Цикл `for`, начинающийся с `for x in count()`, действителен, но работает вечно. Вот пример использования этого бесконечного генератора в цикле `for` с инструкцией `break` для прекращения итераций:

```
for x in count():
    if x > 1000:
        break
    else:
        print(x)
```

Вот более практичная версия генератора-счетчика, которая выдает конечное число значений. Она действует подобно функции `range`, последовательно возвращая значения от заданного в первом аргументе до заданного во втором аргументе:

```
def count(a,b):
    x = a
    while x < b:
        yield x
        x += 1
```

Вызов `count(10,20)` вернет генератор, похожий на `range(10,20)`: мы не сможем получить его значения напрямую, но сможем выполнять итерации по ним, например, в генераторе списка:

```
>>> count(10,20)
<generator object count at 0x0000015FA80EC9A8>
>>> [x for x in count(10,20)]
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Мы можем создавать генераторы-выражения, очень похожие на генераторы списков, заключив код в круглые скобки вместо квадратных. Например, выражение

```
(x*x for x in range(0,10))
```

вернет генератор, вычисляющий квадраты чисел от 0 до 9. Он действует точно так же, как функция-генератор:

```
def squares():
    for x in range(0,10):
        yield x*x
```

Если генератор возвращает конечную последовательность значений, его можно преобразовать в список с помощью функции `list`:

```
>>> list(squares())
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Б.2.4. Кортежи

Кортежи — это итерируемые объекты, во многом похожие на списки, за исключением того, что они не могут изменяться, то есть их нельзя изменить после

создания. Это означает, что кортежи не имеют метода `append`. В частности, после создания кортеж всегда имеет фиксированную длину.

Благодаря этому свойству их удобно использовать для хранения данных, поступающих парами или тройками. Кортежи создаются как списки, с той лишь разницей, что вместо квадратных берутся круглые скобки (или вообще не используются):

```
>>> (1,2)
(1, 2)
>>> ("a","b","c")
('a', 'b', 'c')
>>> 1,2,3,4,5
(1, 2, 3, 4, 5)
```

Если еще раз взглянуть на функцию `zip` из раздела В.2.2, то можно заметить, что возвращаемые ею элементы в действительности являются кортежами. Кортежи в некотором смысле — это наборы по умолчанию в Python. Если написать `a = 1,2,3,4,5` (без круглых скобок), то переменная `a` автоматически будет интерпретироваться как кортеж этих чисел. Аналогично, если завершить функцию инструкцией `return a,b`, то ее результатом будет кортеж `(a,b)`.

Кортежи часто бывают короткими, поэтому обычно не требуется выполнять итерации по ним. К слову, в Python отсутствует такое понятие, как генератор кортежей, но при необходимости можно выполнить обход элементов кортежа, использовав генератор-выражение, и затем преобразовать результат обратно в кортеж с помощью встроенной функции `tuple`. Например, далее показано, как можно реализовать обработку кортежа. На самом деле это генератор-выражение, результат которого передается функции `tuple`:

```
>>> a = 1,2,3,4,5
>>> tuple(x + 10 for x in a)
(11, 12, 13, 14, 15)
```

Б.2.5. Множества

Множества в языке Python — это наборы данных, каждый элемент которых должен быть уникальным, и они не упорядочены. В этой книге множества не нашли особого применения, за исключением преобразования списка в множество для удаления повторяющихся значений. Функция `set` превращает итерируемый объект в множество:

```
>>> dups = [1,2,3,3,3,3,4,5,6,6,6,6,7,8,9,9,9]
>>> set(dups)
{1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> list(set(dups))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Множества в языке Python записываются в виде списков элементов, заключенных в фигурные скобки, что, кстати, совпадает с тем, как записываются множества в математике. Вы можете определить множество с нуля, перечислив некоторые элементы через запятую и заключив их в фигурные скобки. Поскольку множества не являются упорядоченными наборами данных, два множества считаются равными, если имеют одинаковые элементы:

```
>>> set([1,1,2,2,3]) == {3,2,1}
True
```

Б.2.6. Массивы NumPy

Последний тип наборов данных, который мы широко используем в этой книге, не является встроенным набором Python — он реализован в пакете NumPy, который де-факто считается стандартной библиотекой для решения вычислительных задач и предоставляет высокоэффективные реализации вычислительных алгоритмов. Наборы данных этого типа считаются *массивами* NumPy, и не упомянуть о них нельзя из-за повсеместного распространения NumPy. Многие другие библиотеки для Python имеют функции, которые принимают массивы NumPy.

Чтобы задействовать массивы NumPy, необходимо импортировать библиотеку NumPy, а для этого нужно убедиться, что она установлена. Если вы используете Anaconda, как описано в приложении А, то эта библиотека у вас уже есть. В противном случае ее можно установить с помощью диспетчера пакетов `pip`, выполнив команду `pip install numpy` в терминале. После установки NumPy необходимо импортировать ее в программу на Python. Обычно она импортируется с именем `np`:

```
import numpy as np
```

Для создания массива NumPy просто передайте итерируемый объект функции `np.array`:

```
>>> np.array([1,2,3,4,5,6])
array([1, 2, 3, 4, 5, 6])
```

В этой книге мы используем одну из функций NumPy — `np.arange`, которая похожа на встроенную функцию `range`, но работает с числами с плавающей точкой. Если вызвать `np.arange` с двумя аргументами, то она, в отличие от `range`, создаст массив, а не объект диапазона:

```
>>> np.arange(0,10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

При необходимости ей можно передать третий аргумент, который может быть числом с плавающей точкой, определяющим величину шага приращения.

Следующий код создаст массив NumPy со значениями от 0 до 10 с шагом 0,1, всего 100 чисел:

```
>>> np.arange(0,10,0.1)
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. , 1.1, 1.2,
       1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2. , 2.1, 2.2, 2.3, 2.4, 2.5,
       2.6, 2.7, 2.8, 2.9, 3. , 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8,
       3.9, 4. , 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 5. , 5.1,
       5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 6. , 6.1, 6.2, 6.3, 6.4,
       6.5, 6.6, 6.7, 6.8, 6.9, 7. , 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7,
       7.8, 7.9, 8. , 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 9. ,
       9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.8, 9.9])
>>> len(np.arange(0,10,0.1))
100
```

Б.2.7. Словари

Словари — это наборы данных, работающие совершенно иначе, чем списки, кортежи или генераторы. Вместо того чтобы получать доступ к элементам словаря по числовому индексу, их можно пометить другим элементом данных, называемым *ключом*. Чаще всего, по крайней мере в этой книге, на роль ключей выбираются строки. Следующий код определяет словарь `dog` с двумя ключами и соответствующими им значениями, ключ `"name"` связан со строкой `"Melba"`, а ключ `"age"` — с числом 2:

```
dog = {"name" : "Melba", "age" : 2}
```

Для лучшей удобочитаемости в определениях словарей часто используются дополнительные пробелы, а каждая пара «ключ — значение» записывается в отдельной строке. Далее показан тот же словарь `dog` с дополнительными пробелами:

```
dog = {
    "name" : "Melba",
    "age" : 2
}
```

Доступ к значениям в словаре осуществляется с помощью того же синтаксиса, что и при работе со списками, только вместо индекса передается ключ:

```
>>> dog["name"]
'Melba'
>>> dog["age"]
2
```

Чтобы получить все значения из словаря, можно выполнить обход кортежей с парами «ключ — значение», использовав метод `items`. Словари не упорядочивают свои значения, поэтому не следует ожидать, что элементы в выводе будут располагаться в определенном порядке:

```
>>> list(dog.items())
[('name', 'Melba'), ('age', 2)]
```

Б.2.8. Полезные функции для работы с наборами данных

В стандартной библиотеке Python имеется множество полезных встроенных функций для работы с итерируемыми объектами, особенно возвращающими числа. Мы уже видели функцию определения длины `len`, которую будем использовать чаще всего, а также функцию `zip`, но есть и другие, заслуживающие упоминания. Функция `sum` суммирует числовые значения в итерируемом объекте, а функции `max` и `min` возвращают наибольшее и наименьшее значения соответственно:

```
>>> sum([1,2,3])
6
>>> max([1,2,3])
3
>>> min([1,2,3])
1
```

Функция `sorted` возвращает список с отсортированной копией содержимого итерируемого объекта. Важно отметить, что `sorted` возвращает новый список — она не изменяет порядок элементов в исходном списке:

```
>>> q = [3,4,1,2,5]
>>> sorted(q)
[1, 2, 3, 4, 5]
>>> q
[3, 4, 1, 2, 5]
```

Точно так же функция `reversed` возвращает содержимое итерируемого объекта в обратном порядке, оставляя порядок следования элементов в исходном, итерируемом объекте неизменным. Результат — итерируемый объект, а не список, поэтому его нужно преобразовать, чтобы увидеть результат:

```
>>> q
[3, 4, 1, 2, 5]
>>> reversed(q)
<list_reverseiterator at 0x15fb652eb70>
>>> list(reversed(q))
[5, 2, 1, 4, 3]
```

Если потребуется отсортировать или изменить порядок следования элементов на обратный в самом списке, то для этого следует использовать методы `sort` и `reverse`, например, `q.sort()` или `q.reverse()`.

Б.3. РАБОТА С ФУНКЦИЯМИ

Функции в языке Python похожи на мини-программы, которые принимают некоторые входные значения (или, возможно, ничего не принимают), выполняют некоторые вычисления и, возможно, возвращают выходное значение. Мы уже применяли некоторые функции, такие как `math.sqrt` и `zip`, и видели результаты, которые они возвращают для разных входных значений.

Мы можем определять свои функции, используя ключевое слово def. Далее показан пример определения функции square, которая принимает аргумент x, вычисляет значение x * x и сохраняет его в переменной y, а затем возвращает значение y. Подобно циклу for или оператору if, тело функции должно оформляться с отступом, чтобы показать, какие строки принадлежат определению функции:

```
def square(x):
    y = x * x
    return y
```

Результатом этой функции является квадрат входного значения:

```
>>> square(5)
25
```

Этот раздел охватывает некоторые из наиболее продвинутых способов использования функций, описанных в книге.

Б.3.1. Передача функциям нескольких входных данных

Функцию можно определить так, чтобы она принимала столько входных данных, или аргументов, сколько необходимо. Следующая функция принимает три аргумента и складывает их:

```
def add3(x,y,z):
    return x + y + z
```

Иногда полезно, чтобы функция могла принимать переменное количество аргументов. Например, нам может понадобиться функция сложения add, вызов которой add(2,2) возвращает 4, вызов add(1,2,3) возвращает 6 и т. д. Это легко реализовать, добавив звездочку перед именем параметра. Таким параметрам обычно дается имя args. Звездочка указывает на то, что все входные значения должны быть помещены в кортеж args. В теле функции мы можем реализовать логику, которая перебирает все аргументы. Следующая функция add перебирает все переданные ей аргументы и суммирует их, возвращая итог:

```
def add(*args):
    total = 0
    for x in args:
        total += x
    return total
```

Если вызвать эту функцию как add(1,2,3,4,5), она вернет $1 + 2 + 3 + 4 + 5 = 15$, а вызов add() вернет 0. Наша функция add работает не так, как функция sum, упоминавшаяся ранее: sum принимает итерируемый объект, а add — простые значения. Сравните сами:

```
>>> sum([1,2,3,4,5])
15
>>> add(1,2,3,4,5)
15
```

Оператор * имеет еще одно применение — с его помощью можно преобразовать список в набор аргументов функции, например:

```
>>> p = [1,2,3,4,5]
>>> add(*p)
15
```

Этот код эквивалентен вызову `add(1,2,3,4,5)`.

Б.3.2. Именованные аргументы

Использование аргумента со звездочкой — это один из способов получить необязательные параметры. Другой способ — передача *именованных аргументов*. Вот пример функции с двумя необязательными именованными аргументами `name` и `age`, которая возвращает строку, содержащую поздравление с днем рождения:

```
def birthday(name="friend", age=None):
    s = "Happy birthday, %s" % name
    if age:
        s += ", you're %d years old" % age
    return s + "!"
```

(Эта функция использует оператор форматирования строки `%`, который заменяет вхождения `%s` заданной строкой, а вхождения `%d` — заданным числом.) Именованные аргументы `name` и `age` считаются необязательными. По умолчанию аргумент `name` получает значение `"friend"`, поэтому, если вызвать `birthday` без аргументов, она выведет поздравление:

```
>>> birthday()
'Happy birthday, friend!'
```

При желании можно указать конкретное имя. Первый аргумент в вызове этой функции интерпретируется как аргумент `name`, но мы можем явно указать имена аргументов, как показано далее:

```
>>> birthday('Melba')
'Happy birthday, Melba!'
>>> birthday(name='Melba')
'Happy birthday, Melba!'
```

Аргумент `age` тоже считается необязательным и по умолчанию получает значение `None`. В вызове функции можно указать имя (`name`) и возраст (`age`) или только возраст. Поскольку `age` — это второй именованный аргумент в определении

функции, его имя необходимо указывать явно, если функция вызывается без аргумента `name`. Если явно указывать имена аргументов, их можно передавать в любом порядке. Вот несколько примеров:

```
>>> birthday('Melba', 2)
"Happy birthday, Melba, you're 2 years old!"
>>> birthday(age=2)
"Happy birthday, friend, you're 2 years old!"
>>> birthday('Melba', age=2)
"Happy birthday, Melba, you're 2 years old!"
>>> birthday(age=2,name='Melba')
"Happy birthday, Melba, you're 2 years old!"
```

Если аргументов много, то можно упаковать их в словарь и передать его в функцию с помощью оператора **. Он похож на оператор *, но используется для передачи не списка, а словаря с именованными аргументами:

```
>>> dog = {"name" : "Melba", "age" : 2}
>>> dog
{'name': 'Melba', 'age': 2}
>>> birthday(**dog)
"Happy birthday, Melba, you're 2 years old!"
```

Аналогично можно использовать оператор ** в определении функции, чтобы интерпретировать все именованные аргументы, переданные в вызов функции, как единый словарь. Далее показано, как можно переписать функцию `birthday` с помощью оператора **, но в этом случае придется явно указывать имена аргументов при ее вызове:

```
def birthday(**kwargs):
    s = "Happy birthday, %s" % kwargs['name']
    if kwargs['age']:
        s += ", you're %d years old" % kwargs['age']
    return s + "!"
```

Здесь, как видите, вместо переменных `name` и `age` используются `kwargs['name']` и `kwargs['age']`. Вызвать такую функцию можно одним из следующих способов:

```
>>> birthday(**dog)
"Happy birthday, Melba, you're 2 years old!"
>>> birthday(age=2,name='Melba')
"Happy birthday, Melba, you're 2 years old!"
```

Б.3.3. Функции как данные

В языке Python функции интерпретируются как *обычные значения*, то есть можно присваивать их переменным, передавать в функции и возвращать из функций. Другими словами, функции в Python ничем не отличаются от любых других данных. В парадигме *функционального программирования*, которая представлена в главе 4, широко используются функции, которые оперируют другими

функциями. Следующая функция принимает два аргумента, функцию `f` и значение `x` и возвращает значение `f(x)`:

```
def evaluate(f,x):
    return f(x)
```

Если и применить функцию `square` из раздела Б.3, то вызов `evaluate(square,10)` должен вернуть результат вызова `square(10)`, или `100`:

```
>>> evaluate(square,10)
100
```

Более полезной функцией, которая принимает функцию на входе, является встроенная функция `map`. Функция `map` принимает функцию и итерируемый объект и возвращает новый итерируемый объект, полученный путем применения заданной функции к каждому элементу итерируемого объекта. Например, следующий вызов `map` применит функцию `square` к каждому числу, возвращаемому функцией `range(10)`. Преобразовав этот результат в список, можно увидеть первые квадраты первых 10 чисел:

```
>>> map(square,range(10))
<map at 0x15fb752e240>
>>> list(map(square,range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Функции `evaluate` и `map` — это примеры функций, которые принимают другие функции. Функции также могут возвращать другие функции. Например, следующая функция возвращает функцию, которая возводит число в некоторую степень. Обратите внимание на то, что определение функции может целиком находиться внутри другой функции:

```
def make_power_function(power):
    def power_function(x):
        return x ** power

    return power_function
```

Вызов `make_power_function(2)` вернет функцию, которая дает тот же результат, что и функция `square`. Аналогично, вызов `make_power_function(3)` вернет функцию, вычисляющую куб входного значения:

```
>>> square = make_power_function(2)
>>> square(2)
4
>>> cube = make_power_function(3)
>>> cube(2)
8
```

Функция `power_function`, возвращаемая функцией `make_power_function`, запоминает значение переменной `power`, несмотря на то что внутренние переменные функций обычно исчезают бесследно, когда те завершают работу. Функция, запоминающая значения внешних переменных, которые используются в ее определении, называется *замыканием*.

Б.3.4. Лямбда-выражения: анонимные функции

Есть еще одна простая синтаксическая конструкция, которая позволяет создавать функции на лету. С помощью ключевого слова `lambda` можно создать функцию без имени, называемую *анонимной функцией* или *лямбда-выражением*. Это название происходит от греческой буквы λ («лямбда»), которая в теории функционального программирования служит символом определения функций. Чтобы определить функцию в виде лямбда-выражения, нужно вслед за ключевым словом `lambda` перечислить через запятую входные переменные, затем добавить двоеточие и после него — возвращаемое выражение. Вот пример лямбда-выражения, определяющего функцию, которая принимает один аргумент `x` и прибавляет к нему 2:

```
>>> lambda x: x + 2
<function __main__.<lambda>(x)>
```

Лямбда-выражения можно задействовать везде, где можно использовать функции, то есть лямбда выражение можно применить непосредственно к значению:

```
>>> (lambda x: x + 2)(7)
9
```

Вот еще одно лямбда-выражение, принимающее два аргумента и возвращающее сумму значения первого аргумента и удвоенного значения второго аргумента. В следующем примере в первом аргументе передается 2, а во втором — 3, соответственно, результат равен $2 + 2 \cdot 3 = 8$:

```
>>> (lambda x,y: x + 2 * y)(2,3)
8
```

Лямбда-выражение можно присвоить переменной так же, как и любую другую функцию, хотя это несколько противоречит назначению лямбда-выражений играть роль анонимных функций:

```
>>> plus2 = lambda x: x + 2
>>> plus2(5)
7
```

Лямбда-выражения следует использовать с осторожностью, потому что, если функция делает что-то интересное, она, вероятно, заслуживает того, чтобы дать ей имя. Одно из мест, где можно применять лямбда-выражения, — в функциях, возвращающих другие функции. Например, функцию `make_power_function` можно реализовать с помощью лямбда-выражения, как показано далее:

```
def make_power_function(p):
    return lambda x: x ** p
```

Эта функция действует точно так же, как ее предыдущая версия:

```
>>> make_power_function(2)(3)
9
```

Имя вмещающей функции объясняет назначение возвращаемой функции, поэтому от определения имени возвращаемой функции не так уж много пользы. Лямбда-выражения можно также применять в качестве аргументов функций. Например, вот как можно кратко реализовать увеличение на 2 каждого числа от 0 до 9:

```
map(lambda x: x + 2, range(0,9))
```

Чтобы увидеть получившуюся последовательность, нужно преобразовать этот результат в список. Однако в большинстве случаев предпочтительнее использовать синтаксис генераторов списков. Вот эквивалентное решение с генератором списка:

```
[x+2 for x in range(0,9)]
```

Б.3.5. Применение функций к массивам NumPy

В библиотеке NumPy есть свои версии встроенных математических функций, которые можно применять сразу ко всем элементам массива NumPy. Например, `np.sqrt` — это функция извлечения квадратного корня, которая может извлекать квадратный корень из числа или из всех элементов массива NumPy. Так, `np.sqrt(np.arange(0,10))` возвращает массив NumPy с квадратными корнями целых чисел от 0 до 9:

```
>>> np.sqrt(np.arange(0,10))
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
       2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ])
```

Это не просто функция, созданная для удобства. На самом деле реализация в NumPy работает быстрее, чем перебор массива в Python. Если вам понадобится применить пользовательскую функцию к каждому элементу массива NumPy, то задействуйте функцию `np.vectorize`. Вот пример функции, принимающей одно число и возвращающей другое:

```
def my_function(x):
    if x % 2 == 0:
        return x/2
    else:
        return 0
```

Следующий код векторизует функцию и применяет ее к каждому элементу массива NumPy `np.arange(0,10)`:

```
>>> my_numpy_function = np.vectorize(my_function)
>>> my_numpy_function(np.arange(0,10))
array([0., 0., 1., 0., 2., 0., 3., 0., 4., 0.])
```

Б.4. ДАННЫЕ С ПЛАВАЮЩЕЙ ТОЧКОЙ И MATPLOTLIB

Matplotlib — самая популярная библиотека для построения графиков в Python. С ее помощью на протяжении всей книги создавались графики наборов данных, функций и рисунков геометрических фигур. Чтобы избежать обсуждения конкретных библиотек, я скрыл большую часть деталей использования Matplotlib в функциях-обертках, поэтому вы можете задействовать их для выполнения всех упражнений и мини-проектов. Для тех, кому интересны детали реализации, в этом разделе я дам краткий обзор приемов создания графиков с помощью Matplotlib. При установке дистрибутива Anaconda библиотека Matplotlib устанавливается автоматически, в противном случае ее можно установить вручную командой `pip install matplotlib`.

Б.4.1. Создание диаграммы рассеяния

Диаграммы рассеяния удобно использовать для визуализации наборов упорядоченных пар чисел (x, y) в виде точек на плоскости (более подробно они рассматриваются в главе 2). Чтобы иметь возможность создавать диаграммы рассеяния или любые другие с помощью Matplotlib, первым делом необходимо установить библиотеку и импортировать ее в сценарий на Python. Обычно модуль `pyplot` из библиотеки Matplotlib импортируется под именем `plt`:

```
import matplotlib.pyplot as plt
```

Допустим, что мы решили построить диаграмму рассеяния для точек (1, 1), (2, 4), (3, 9), (4, 16) и (5, 25), которые являются парами некоторых чисел и их квадратов (рис. Б.1). Если представить их как точки с координатами (x, y), то значения x будут равны 1, 2, 3, 4 и 5, а значения y — 1, 4, 9, 16 и 25. Чтобы построить диаграмму рассеяния, используется функция `plt.scatter`, которой в первом аргументе передается список значений x, а во втором — список значений y:

```
x_values = [1,2,3,4,5]
y_values = [1,4,9,16,25]
plt.scatter(x_values,y_values)
```

Значения x точек определяют их позиции по горизонтали, а значения y — позиции по вертикали. Обратите внимание на то, что Matplotlib автоматически масштабирует область графика, чтобы уместить все точки, поэтому в данном случае оси x и y имеют разный масштаб.

Функция `plt.scatter` имеет также несколько именованных аргументов, которые можно использовать для настройки внешнего вида диаграммы (рис. Б.2). Например, аргумент `marker` задает форму точек на графике, а аргумент `c` — их цвет. Следующая инструкция отображает те же данные красными крестиками вместо синих кружков по умолчанию:

```
plt.scatter(x_values,y_values,marker='x',c='red')
```

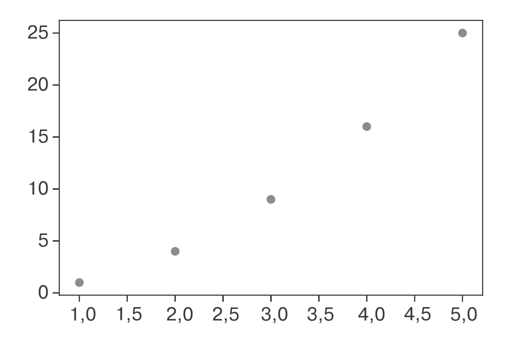


Рис. Б.1. Диаграмма рассеяния, созданная с помощью функции plt. scatter из библиотеки Matplotlib

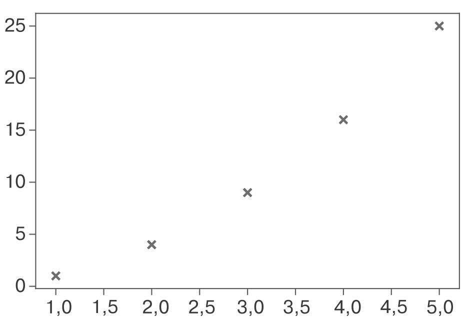


Рис. Б.2. Настройка внешнего вида диаграммы рассеяния

Описание всех именованных аргументов настроек, поддерживаемых библиотекой Matplotlib, можно найти в документации по адресу: https://matplotlib.org/.

Б.4.2. Создание линейной диаграммы

Если вызвать функцию `plt.plot` вместо `plt.scatter`, то заданные точки будут соединены отрезками. Такие диаграммы иногда называют *линейными* (рис. Б.3), например:

```
plt.plot(x_values,y_values)
```

Одно из полезных применений этой функции — ее вызов с двумя точками, чтобы нарисовать прямую. Например, можно написать функцию, которая принимает две точки (x, y) как кортежи и рисует соединяющий их отрезок прямой (рис. Б.4), извлекая значения x и y и вызывая `plt.plot`:

```
def plot_segment(p1,p2):
    x1,y1 = p1
    x2,y2 = p2
    plt.plot([x1,x2],[y1,y2],marker='o')
```

Этот пример показывает также возможность использования именованного аргумента `marker` для того, чтобы отметить отдельные точки в дополнение к рисованию линии:

```
point1 = (0,3)
point2 = (2,1)
plot_segment(point1,point2)
```

Функция `draw_segment` может служить примером функции-обертки: теперь вы сможете использовать `draw_segment`, когда понадобится нарисовать отрезок между двумя точками (x, y), а не брать функции из Matplotlib напрямую.

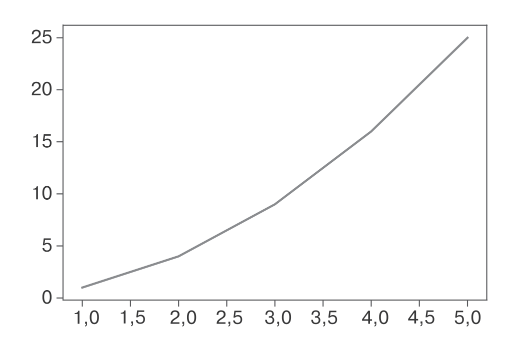


Рис. Б.3. Создание линейной диаграммы с помощью функции plt.plot из библиотеки Matplotlib

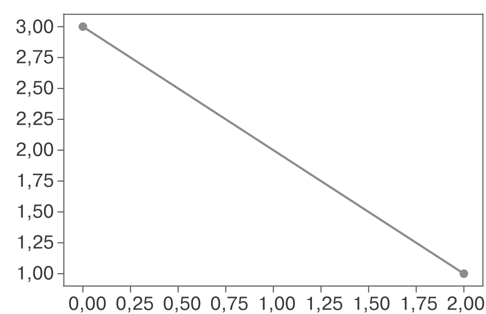


Рис. Б.4. Функция рисует отрезок прямой, соединяющий две точки

Еще одно важное применение линейной диаграммы — построение *графика* функции, то есть рисование всех пар $(x, f(x))$ для некоторой фиксированной функции f в некотором диапазоне значений x. Теоретически гладкий непрерывный график состоит из бесконечного множества точек (рис. Б.5). Мы не можем использовать бесконечно много точек, но чем больше их будет, тем точнее выглядит график. Вот график $f(x) = \sin(x)$ от $x = 0$ до $x = 10$, нарисованный с помощью 1000 точек:

```
x_values = np.arange(0,10,0.01)
y_values = np.sin(x_values)
plt.plot(x_values,y_values)
```

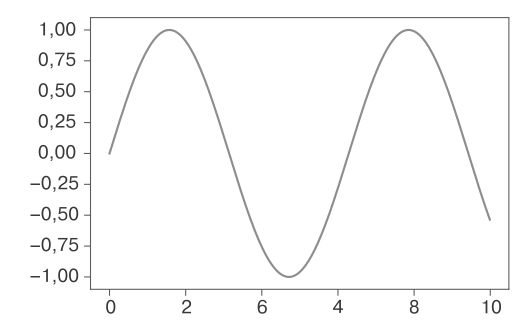


Рис. Б.5. Используя большое количество точек, можно получить довольно гладкий график функции

Б.4.3. Дополнительные настройки диаграмм

Как я уже упоминал, лучший способ узнать больше о настройке диаграмм в Matplotlib — заглянуть в документацию на сайте matplotlib.org. Тем не менее мне хотелось бы упомянуть здесь несколько важных способов управления внешним видом диаграмм, которые часто встречаются в примерах в этой книге.

Первый — установка масштаба и размера диаграммы. Вы могли заметить, что диаграмма, полученная вызовом `plot_segment(point1,point2)`, нарисована с нарушением пропорций. Чтобы сохранить пропорции диаграммы, нужно явно установить одинаковые границы по осям x и y (рис. Б.6). Например, следующий пример устанавливает границы x и y в диапазоне от 0 до 5:

```
plt.ylim(0,5)
plt.xlim(0,5)
plot_segment(point1,point2)
```

Но пропорции все еще несколько нарушены. Визуально единица на оси x больше единицы на оси y. Чтобы соблюсти визуальные пропорции, нужно сделать график квадратным с помощью метода `set_size_inches`. Этот метод принадлежит объекту фигуры, с которым работает Matplotlib. Получить его можно с помощью метода `gcf` (get current figure — получить текущую фигуру) модуля `plt`. Следующий код рисует отрезок прямой с правильными пропорциями в области размером 5×5 дюймов. В зависимости от особенностей дисплея фактические размеры могут не совпадать с заданными, но пропорции теперь должны быть правильными (рис. Б.7):

```
plt.ylim(0,5)
plt.xlim(0,5)
plt.gcf().set_size_inches(5,5)
plot_segment(point1,point2)
```

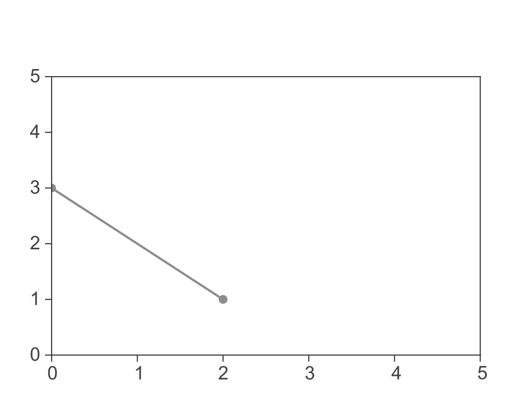


Рис. Б.6. Вывод отрезка прямой с сохранением пропорций на диаграмме путем задания границ по осям *x* и *y*

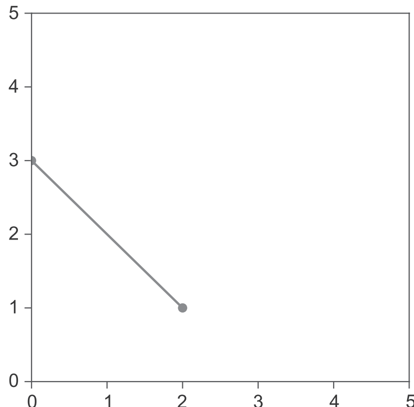


Рис. Б.7. Вывод отрезка прямой с визуально правильными пропорциями путем задания размеров фигуры в дюймах

Другая важная настройка — установка меток для осей и всего графика. Например, с помощью функции `plt.title` можно добавить заголовок в текущий график,

а с помощью функций `plt.xlabel` и `plt.ylabel` — метки для осей x и y соответственно (рис. Б.8). Вот пример добавления меток в график функции синуса:

```
x_values = np.arange(0,10,0.01)
y_values = np.sin(x_values)
plt.plot(x_values,y_values)
plt.title('Graph of sin(x) vs. x',fontsize=16)
plt.xlabel('this is the x value',fontsize=16)
plt.ylabel('the value of sin(x)',fontsize=16)
```

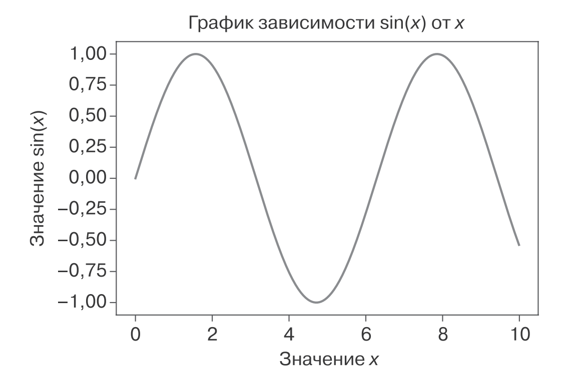


Рис. Б.8. График с заголовком и метками осей

Б.5. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА PYTHON

Парадигма объектно-ориентированного программирования (ООП) делает упор на организацию данных программы в *классы*. Классы могут хранить значения, называемые *свойствами*, а также функции, называемые *методами*, и тем самым связывать вместе данные и функциональность. Вам не нужно владеть навыками ООП, чтобы оценить эту книгу, но некоторые математические концепции имеют объектно-ориентированный оттенок, особенно в главах 6 и 10, где используются классы и некоторые принципы объектно-ориентированного проектирования, помогающие понять математику. В этом разделе я дам краткое введение в классы и ООП на языке Python.

Б.5.1. Определение классов

Рассмотрим конкретный пример. Предположим, вы пишете программу на Python, работающую с геометрическими фигурами, например, приложение для рисования. Одна из фигур, которые вы, возможно, захотите описать, — это

прямоугольник. Чтобы описать прямоугольник, мы можем определить класс Rectangle, описывающий свойства прямоугольников, а затем создать экземпляры этого класса, представляющие определенные прямоугольники.

В языке Python класс определяется с помощью ключевого слова class, а имя класса обычно пишется с прописной буквы, например Rectangle. Строки с отступом под именем класса описывают свойства (значения) и методы (функции) класса. Самый простой метод класса — это его *конструктор*, то есть функция, позволяющая создавать экземпляры класса. В Python конструкторы имеют специальное имя __init__. Прямоугольник можно описать двумя числами, представляющими высоту и ширину. В данном случае функция __init__ принимает три значения: первое представляет создаваемый экземпляр класса, а два других — значения высоты и ширины. Задача конструктора — инициализировать свойства высоты и ширины нового экземпляра в соответствии с аргументами:

```
class Rectangle():
    def __init__(self,w,h):
        self.width = w
        self.height = h
```

Определив конструктор, мы можем использовать имя класса как функцию, которая принимает два числа и возвращает объект Rectangle. Например, Rectangle(3,4) создаст экземпляр со свойством ширины width, равным 3, и свойством высоты height, равным 4. Несмотря на то что в определении конструктора присутствует аргумент self, его не нужно указывать при вызове конструктора. Создав объект Rectangle, можем получить значения его высоты и ширины:

```
>>> r = Rectangle(3,4)
>>> type(r)
__main__.Rectangle
>>> r.width
3
>>> r.height
4
```

Б.5.2. Определение методов

Метод — это функция, связанная с классом, которая может выполнять вычисления, связанные с экземпляром, или придавать экземплярам какую-то функциональную возможность. Для прямоугольника имел бы смысл метод area(), вычисляющий площадь, то есть произведение высоты на ширину. Подобно конструктору, любой метод должен принимать параметр self, представляющий текущий экземпляр. И снова, вызывая метод, вам не нужно передавать ему

параметр `self` — ему автоматически присваивается ссылка на текущий объект, для которого вызывается метод:

```
class Rectangle():
    def __init__(self,w,h):
        self.width = w
        self.height = h

    def area(self):
        return self.width * self.height
```

Теперь, чтобы найти площадь прямоугольника, мы можем вызвать его метод `area`:

```
>>> Rectangle(3,4).area()
12
```

Обратите внимание на то, что в функцию не передается аргумент `self` — он автоматически будет инициализирован ссылкой на экземпляр `Rectangle(3,4)`. Еще в прямоугольнике не помешал бы метод `scale`, принимающий число и возвращающий новый объект `Rectangle` со значениями высоты и ширины оригинала, умноженными на это число (далее я буду использовать многоточия «...», обозначая уже написанный код в классе `Rectangle`):

```
class Rectangle():
    ...
    def scale(self, factor):
        return Rectangle(factor * self.width, factor * self.height)
```

В следующем примере вызов `Rectangle(2,1)` создаст прямоугольник с шириной 2 и высотой 1. Если масштабировать его в 3 раза, получится новый прямоугольник с шириной 6 и высотой 3:

```
>>> r = Rectangle(2,1)
>>> s = r.scale(3)
>>> s.width
6
>>> s.height
3
```

Б.5.3. Специальные методы

Некоторые методы могут быть доступны автоматически или иметь особый эффект, если их реализовать. Например, метод `__dict__` доступен по умолчанию для каждого экземпляра нового класса. Он возвращает словарь со всеми свойствами экземпляра. Без дополнительных модификаций класса `Rectangle` мы можем написать такой код:

```
>>> Rectangle(2,1).__dict__
{'width': 2, 'height': 1}
```

Другое имя специального метода — `__eq__`. Будучи реализованным, он описывает поведение оператора `==` для экземпляров класса и, следовательно, определяет равенство или неравенство двух экземпляров. Без реализации этого специального метода разные экземпляры класса всегда считаются неравными, даже если содержат одни и те же данные:

```
>>> Rectangle(3,4) == Rectangle(3,4)
False
```

О прямоугольниках можно сказать, что они равны, если имеют одинаковые геометрические размеры, ширину и высоту. Соответственно, мы можем реализовать это специальное поведение, определив метод `__eq__`. Метод `__eq__` принимает два аргумента: аргумент `self`, как обычно, и второй аргумент, представляющий другой экземпляр, с которым сравнивается текущий экземпляр `self`:

```
class Rectangle():
    ...
    def __eq__(self,other):
        return self.width == other.width and self.height == other.height
```

Теперь экземпляры `Rectangle` будут считаться равными, если имеют одинаковые высоту и ширину:

```
>>> Rectangle(3,4) == Rectangle(3,4)
True
```

Еще один полезный специальный метод — это `__repr__`, который выводит строковое представление объекта. Следующая реализация метода `__repr__` помогает увидеть ширину и высоту прямоугольника с первого взгляда:

```
class Rectangle():
    ...
    def __repr__(self):
        return 'Rectangle (%r by %r)' % (self.width, self.height)
```

Посмотрим, как он работает:

```
>>> Rectangle(3,4)
Rectangle (3 by 4)
```

Б.5.4. Перегрузка операторов

Есть множество специальных методов, которые можно реализовать, чтобы гарантировать особое поведение операторов Python при работе с экземплярами класса. Перепрофилирование операторов, которые имеют смысл в процессе работы с объектами нового класса, называется *перегрузкой операторов*. Например, методы `__mul__` и `__rmul__` описывают поведение оператора умножения * в ходе работы с экземплярами класса, стоящими справа и слева от оператора

соответственно. Имея экземпляр `Rectangle r`, мы могли бы написать `r * 3` или `3 * r`, чтобы выполнить масштабирование прямоугольника в 3 раза. Следующие реализации `__mul__` и `__rmul__` вызывают метод `scale`, который мы уже определили, и создают новый прямоугольник, масштабированный в соответствии с указанным коэффициентом:

```
class Rectangle():
    ...
    def __mul__(self,factor):
        return self.scale(factor)

    def __rmul__(self,factor):
        return self.scale(factor)
```

Нетрудно убедиться в том, что выражения `10 * Rectangle(1,2)` и `Rectangle(1,2) * 10` вернут новые экземпляры `Rectangle` шириной 10 и высотой 20:

```
>>> 10 * Rectangle(1,2)
Rectangle (10 by 20)
>>> Rectangle(1,2) * 10
Rectangle (10 by 20)
```

Б.5.5. Методы класса

Методы — это функции, которые можно вызывать только для существующих экземпляров класса. Однако есть другая разновидность методов — *методы класса*, которые представляют функции, связанные с самим классом, а не с конкретным экземпляром. Для `Rectangle` метод класса может содержать некоторую функциональность, имеющую отношение к прямоугольникам в целом, а не к конкретному прямоугольнику.

Одно из типичных применений методов класса — это создание альтернативного конструктора. Например, мы можем определить метод класса для класса `Rectangle`, принимающий одно число и возвращающий прямоугольник с высотой и шириной, равными этому числу. Другими словами, этот метод класса строит квадрат с заданной длиной стороны. Первый аргумент метода класса представляет сам класс, его имя часто сокращается до `cls`:

```
class Rectangle():
    ...
    @classmethod
    def square(cls,side):
        return Rectangle(side,side)
```

Определив этот метод класса, мы можем выполнить вызов `Rectangle.square(5)` и получить тот же результат, что и при вызове `Rectangle(5,5)`.

Б.5.6. Наследование и абстрактные классы

Последняя тема ООП, которую мы затронем, — это *наследование*. Если мы говорим, что класс `A` наследует класс `B`, это все равно что сказать, что экземпляры класса `A` являются частными случаями класса `B`: они подобны экземплярам класса `B`, но имеют некоторые дополнительные или измененные функции. В этом случае мы также говорим, что `A` является *подклассом* `B` или что `B` является *надклассом* (*суперклассом*) `A`. В качестве простого примера создадим подкласс `Square`, наследующий класс `Rectangle` и представляющий квадраты, сохраняя при этом большую часть базовой логики класса `Rectangle`. Объявление `class Square(Rectangle)` означает, что `Square` — это подкласс `Rectangle`, а вызов `super().__init__` запускает конструктор суперкласса (`Rectangle`) из конструктора `Square`:

```
class Square(Rectangle):
    def __init__(self,s):
        return super().__init__(s,s)

    def __repr__(self):
        return "Square (%r)" % self.width
```

Этого достаточно, чтобы определить класс `Square`, и теперь мы можем применить любой метод `Rectangle` к экземпляру `Square`:

```
>> Square(5).area()
25
```

Иногда бывает желательно реализовать по-иному, или *переопределить*, некоторые методы, такие как `scale`, который по умолчанию возвращает масштабированный квадрат в виде экземпляра `Rectangle`.

В ООП широко используется общий шаблон проектирования, когда два класса наследуют один *абстрактный базовый класс* — класс, определяющий некоторые общие методы или код, но который нельзя применять для создания экземпляров. В качестве примера предположим, что у нас есть аналогичный класс `Circle`, представляющий окружность заданного радиуса. Бóльшая часть реализации класса `Circle` аналогична классу `Rectangle`:

```
from math import pi

class Circle():
    def __init__(self, r):
        self.radius = r

    def area(self):
        return pi * self.radius * self.radius
```

```
    def scale(self, factor):
        return Circle(factor * self.radius)

    def __eq__(self,other):
        return self.radius == other.radius

    def __repr__(self):
        return 'Circle (radius %r)' % self.radius

    def __mul__(self,factor):
        return self.scale(factor)

    def __rmul__(self,factor):
        return self.scale(factor)
```

(Как мы знаем из школьного курса геометрии, площадь круга радиусом *r* равна πr^2.) Если в программе используется несколько разных фигур, то можно унаследовать классы `Circle` и `Rectangle` от общего класса `Shape`. Понятие фигуры (shape) недостаточно конкретно, чтобы создавать экземпляры этого класса, поэтому в нем можно было бы реализовать лишь некоторые из методов, а остальные отметить как *абстрактные методы*, то есть такие, которые нельзя реализовать для `Shape`, но можно для любого другого конкретного подкласса.

Следующий пример иллюстрирует создание абстрактного класса в Python. ABC означает abstract base class (абстрактный базовый класс), а `ABC` — это специальный базовый класс, который должны наследовать все абстрактные классы в Python:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def scale(self, factor):
        pass

    def __eq__(self,other):
        return self.__dict__ == other.__dict__

    def __mul__(self,factor):
        return self.scale(factor)

    def __rmul__(self,factor):
        return self.scale(factor)
```

Перегруженные операторы равенства и умножения реализованы полностью, причем `__eq__` проверяет совпадение всех свойств двух фигур. Реализация методов вычисления площади и создания масштабного экземпляра оставлены за дочерними классами, потому что их реализации зависят от конкретной фигуры.

Если бы мы решили повторно определить класс Rectangle на основе абстрактного базового класса Shape, то могли бы начать с того, что унаследовали бы в нем класс Shape, но реализовали собственный конструктор:

```
class Rectangle(Shape):
    def __init__(self,w,h):
        self.width = w
        self.height = h
```

При попытке создать экземпляр Rectangle, имея только этот код, мы получили бы ошибку, сообщающую об отсутствии реализаций методов area и scale:

```
>>> Rectangle(1,3)
TypeError: Can't instantiate abstract class Rectangle with abstract methods
area, scale
```

Чтобы исправить проблему, можно включить предыдущие реализации этих методов:

```
class Rectangle(Shape):
    def __init__(self,w,h):
        self.width = w
        self.height = h

    def area(self):
        return self.width * self.height

    def scale(self, factor):
        return Rectangle(factor * self.width, factor * self.height)
```

После добавления методов, реализующих поведение, специфическое для прямоугольника, мы сможем создать его экземпляр и использовать все функции базового класса Shape. Например, операторы равенства и умножения будут вести себя в точности, как ожидается:

```
>>> 3 * Rectangle(1,2) == Rectangle(3,6)
True
```

Теперь мы можем быстро реализовать класс Circle, Triangle и любой другой двухмерной фигуры, которые будут иметь конкретные методы area и scale и общие перегруженные операторы.

Приложение B
Загрузка и отображение трехмерных моделей с помощью OpenGL и PyGame

В главе 3 и далее, где мы начинаем писать программы для преобразования и анимации графики, я использую библиотеки OpenGL и PyGame вместо Matplotlib. В этом приложении кратко описывается, как организовать игровой цикл в PyGame и отображать трехмерные модели в последовательных кадрах. Кульминацией приложения станет реализация функции `draw_model`, реализующей изображение трехмерной модели, похожей на чайник, с которым мы работали в главе 4.

Целью `draw_model` является инкапсуляция (сокрытие) деталей работы с библиотекой, поэтому вам не придется тратить много времени на борьбу с OpenGL. Но если хотите понять, как работает эта функция, то опробуйте примеры, приведенные в приложении, и экспериментируйте с кодом самостоятельно. Начнем с того, что воссоздадим октаэдр из главы 3 с помощью PyOpenGL (интерфейсной библиотеки поддержки OpenGL в Python) и PyGame.

B.1. ВОССОЗДАНИЕ ОКТАЭДРА ИЗ ГЛАВЫ 3

Чтобы начать работу с библиотеками PyOpenGL и PyGame, их необходимо установить. Я рекомендую использовать `pip`:

```
> pip install PyGame
> pip install PyOpenGL
```

Для начала покажу, как задействовать эти библиотеки для воссоздания уже проделанной нами работы — отображения простого трехмерного объекта.

Файл `octahedron.py`, который вы найдете в примерах исходного кода для приложения В, начинается с инструкций импорта. Первые несколько инструкций импортируют две новые библиотеки, PyGame и PyOpenGL, а остальные должны быть вам знакомы по главе 3. В частности, продолжим использовать уже созданные нами арифметические функции поддержки трехмерных векторов, собранные в файле `vectors.py` (также имеется в примерах исходного кода для книги). Вот эти инструкции импорта:

```
import pygame
from pygame.locals import *
from OpenGL.GL import *
from OpenGL.GLU import *
import matplotlib.cm
from vectors import *
from math import *
```

В библиотеке OpenGL имеются средства автоматического отображения теней, но мы продолжим применять механизм, рассмотренный в главе 3. Для отображения затененных сторон октаэдра можно использовать синюю палитру из Matplotlib:

```
def normal(face):
    return(cross(subtract(face[1], face[0]), subtract(face[2], face[0])))

blues = matplotlib.cm.get_cmap('Blues')

def shade(face,color_map=blues,light=(1,2,3)):
    return color_map(1 - dot(unit(normal(face)), unit(light)))
```

Далее нужно задать геометрию октаэдра и характеристики источника света. И снова возьмем код из главы 3:

```
light = (1,2,3)
faces = [
    [(1,0,0), (0,1,0), (0,0,1)],
    [(1,0,0), (0,0,-1), (0,1,0)],
    [(1,0,0), (0,0,1), (0,-1,0)],
    [(1,0,0), (0,-1,0), (0,0,-1)],
    [(-1,0,0), (0,0,1), (0,1,0)],
    [(-1,0,0), (0,1,0), (0,0,-1)],
    [(-1,0,0), (0,-1,0), (0,0,1)],
    [(-1,0,0), (0,0,-1), (0,-1,0)],
]
```

Далее мы вступаем на пока незнакомую территорию. Отобразим октаэдр в игровом окне PyGame, для создания которого требуется написать несколько шаблонных строк кода, как показано далее. Эти строки инициализируют игровой движок, устанавливают размер окна в пикселах и настраивают PyGame на использование OpenGL в качестве графического движка:

```
pygame.init()
display = (400,400)
window = pygame.display.set_mode(display,
                                 DOUBLEBUF|OPENGL)
```

Отображать графику в окне размером 400 × 400 пикселов

PyGame с использованием OpenGL и встроенной оптимизации, которая называется двойной буферизацией (ее понимание неважно для нашей цели)

В упрощенном примере в разделе 3.5 октаэдр нарисован с точки зрения человека, находящегося выше на оси *z* и глядящего вниз. Мы вычислили, какие треугольники должны быть видны наблюдателю, и спроецировали их на двухмерную плоскость, удалив ось *z*. OpenGL имеет встроенные функции для более точной настройки перспективы:

```
gluPerspective(45, 1, 0.1, 50.0)
glTranslatef(0.0,0.0, -5)
glEnable(GL_CULL_FACE)
glEnable(GL_DEPTH_TEST)
glCullFace(GL_BACK)
```

Для изучения математики не требуется знать, что делают эти функции, но кратко объясню для тех, кому интересно. Вызов `gluPerspective` описывает перспективу с точки зрения глядящего на сцену. В данном случае он определяет угол обзора 45° и отношение сторон 1:1. Это означает, что единицы измерения по вертикали и по горизонтали отображаются одинаково. Числа 0,1 и 50 накладывают ограничения на видимые координаты *z*: никакие объекты, расположенные дальше 50 единиц или ближе 0,1 единиц от наблюдателя, не будут отображаться. Это сделано для оптимизации производительности. Вызов `glTranslatef` сообщает, что наблюдатель находится над сценой на высоте 5 единиц выше по оси *z*, то есть вся сцена должна перемещаться вниз на вектор (0, 0, –5). Вызов `glEnable(GL_CULL_FACE)` включает параметр OpenGL, который автоматически скрывает многоугольники, лицевой стороной направленные от зрителя, что избавляет нас от дополнительной работы, которую мы проделали в главе 3, а вызов `glEnable(GL_DEPTH_TEST)` гарантирует, что многоугольники, находящиеся ближе к нам, будут отображаться поверх более отдаленных многоугольников. Наконец, вызов `glCullFace(GL_BACK)` включает параметр OpenGL, который автоматически скрывает многоугольники, обращенные к нам, но находящиеся позади других многоугольников. При отображении сферы эта проблема не проявлялась, но может проявиться при отображении более сложных форм.

Наконец, реализуем основной код, рисующий октаэдр. Поскольку наша конечная цель — это анимация объектов, напишем код, который рисует объект в цикле, в результате получится некоторое подобие кадров в фильме, показывающих один и тот же октаэдр во времени. И как любое видео любого покоящегося объекта, результат неотличим от статической картинки.

Чтобы отобразить один кадр, переберем все векторы, определим, как их затенить, нарисуем их с помощью OpenGL и обновим кадр с помощью PyGame. Внутри

бесконечного цикла `while` этот процесс может повторяться настолько быстро, насколько возможно:

```
clock = pygame.time.Clock()          ← Инициализировать часы в PyGame
                                       для измерения хода времени
while True:
    for event in pygame.event.get():   ← В каждой итерации проверить события,
        if event.type == pygame.QUIT:    полученные PyGame, и завершить работу,
            pygame.quit()                если пользователь закрыл окно
            quit()

    clock.tick()                       ← Обновить значение часов
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
    glBegin(GL_TRIANGLES)              ← Сообщить OpenGL, что мы начинаем
                                         рисовать треугольники
    for face in faces:
        color = shade(face,blues,light)
        for vertex in face:
            glColor3fv((color[0],
                        color[1],
                        color[2]))     ← Для каждой вершины каждой грани (треугольника)
                                         задать цвет с учетом затенения
            glVertex3fv(vertex)        ← Определить следующую вершину
                                         текущего треугольника
    glEnd()
    pygame.display.flip()              ← Сообщить PyGame, что новый кадр
                                         готов и его можно отобразить
```

После запуска этого кода на экране появится окно PyGame размером 400 × 400 пикселов с изображением октаэдра, напоминающим изображение из главы 3 (рис. В.1).



Рис. В.1. Октаэдр, отображаемый в окне PyGame

Чтобы показать, что это не статическая картинка, а своеобразный фильм, в котором одинаковые кадры сменяют друг друга, включите в конец цикла `while True` следующую строку:

```
print(clock.get_fps())
```

Она выведет мгновенное измеренное значение скорости (в кадрах в секунду), с которой PyGame снова и снова отображает октаэдр. Для такой простой анимации PyGame должна достичь или превысить максимальную частоту кадров по умолчанию, равную 60 кадрам в секунду.

Но какой смысл отображать столько кадров, если ничего не меняется? Потерпите, как только мы добавим векторное преобразование в каждый кадр, октаэдр начнет двигаться. На данный момент мы можем схитрить, смещая «камеру» с каждым кадром вместо перемещения октаэдра.

В.2. ИЗМЕНЕНИЕ ТОЧКИ ЗРЕНИЯ

Функция `glTranslatef` из предыдущего раздела сообщает библиотеке OpenGL позицию, с которой мы наблюдаем трехмерную сцену. Точно так же функция `glRotatef` позволяет изменить угол, под которым мы это делаем. Вызов `glRotatef(theta, x, y, z)` поворачивает всю сцену на угол `theta` вокруг оси, заданной вектором (x, y, z).

Поясню, что я имею в виду под поворотом на угол вокруг оси. Вспомните знакомый пример — вращение Земли в космосе. Земля поворачивается на 360° за сутки, или на 15° за час. *Ось* — это невидимая линия, вокруг которой вращается Земля, она проходит через Северный и Южный полюса — единственные две точки, которые не вращаются вокруг оси. Ось вращения Земли направлена не строго вертикально, а наклонена на 23,5° (рис. В.2).

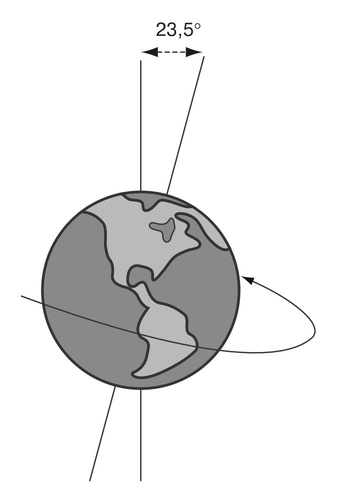


Рис. В.2. Пример объекта, вращающегося вокруг оси. Ось вращения Земли наклонена на 23,5° относительно плоскости ее орбиты

Вектор (0, 0, 1) направлен вдоль оси z, поэтому вызов `glRotatef(30,0,0,1)` повернет сцену на угол 30° вокруг оси z. Вызов `glRotatef(30,0,1,1)` тоже повернет сцену на 30°, но вокруг оси (0, 1, 1), которая наклонена на 45° между осями координат y и z. Если выполнить вызов `glRotatef(30,0,0,1)` или `glRotatef(30,0,1,1)` после `glTranslatef(...)`, то мы увидим повернутый октаэдр (рис. В.3).

Обратите внимание на то, что освещенность четырех видимых граней октаэдра не изменилась. Это объясняется тем, что не изменился ни один из векторов — координаты вершин октаэдра и источника света остались прежними! Мы лишь изменили положение «камеры» относительно октаэдра. Когда изменим положение самого октаэдра, то также увидим, как меняется его освещенность.

Чтобы получить анимационный эффект вращения октаэдра, можно в каждом кадре вызывать функцию `glRotate` и передавать ей небольшой угол. Например, если принять, что PyGame рисует октаэдр со скоростью около 60 кадров в секунду и мы будем выполнять вызов `glRotatef (1, x, y, z)` в каждом кадре, то каждую секунду октаэдр будет поворачиваться примерно на 60° вокруг оси (x, y, z). Добавление вызова `glRotatef(1,1,1,1)` в бесконечный цикл `while` перед вызовом `glBegin` создаст эффект поворота октаэдра на 1° в каждом кадре вокруг оси, указывающей в направлении (1, 1, 1), как показано на рис. В.4.

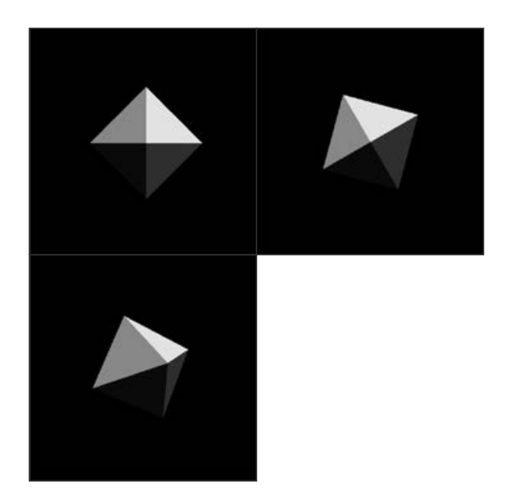


Рис. В.3. Вид октаэдра с трех разных точек зрения, повернутых с использованием функции glRotatef из OpenGL

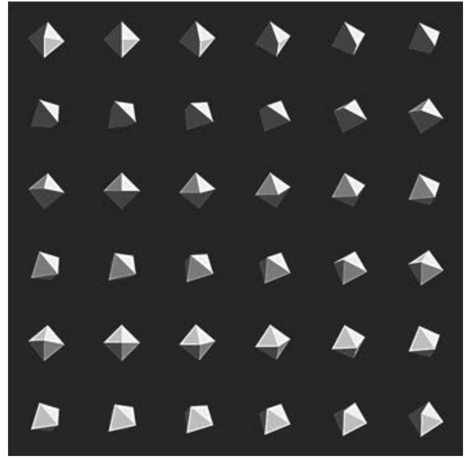


Рис. В.4. Каждый десятый кадр из полученного анимационного эффекта с поворотом октаэдра на 1° за кадр. Октаэдр совершает полный оборот за 360 кадров

Эта скорость вращения будет выдерживаться точно, только если PyGame отображает кадры с частотой ровно 60 кадров в секунду. Однако в общем случае это может оказаться неверным: если для вычисления всех векторов и отрисовки всех многоугольников в сложной сцене потребуется больше 1/60 с, то движение замедлится. Чтобы сохранить движение сцены постоянным независимо от частоты кадров, можно использовать часы PyGame.

Допустим, нам нужно, чтобы сцена делала полный оборот (360°) каждые 5 секунд. Часы PyGame измеряют время в миллисекундах — тысячных долях секунды. Чтобы получить угол поворота за тысячную долю секунды, нужно угол поворота за секунду разделить на 1000:

```
degrees_per_second = 360./5
degrees_per_millisecond = degrees_per_second / 1000
```

Созданные нами часы PyGame имеют метод `tick()`, который переводит часы вперед и возвращает количество миллисекунд, прошедших с момента предыдущего вызова `tick()`. Это позволяет узнать, сколько миллисекунд прошло с момента отображения предыдущего кадра, и вычислить угол, на который сцена должна была повернуться за это время:

```
milliseconds = clock.tick()
glRotatef(milliseconds * degrees_per_millisecond, 1,1,1)
```

Такой вызов `glRotatef` в каждом кадре гарантирует, что сцена будет поворачиваться ровно на 360° каждые 5 секунд. В файле *rotate_octahedron.py* в примерах с исходным кодом для приложения С вы сможете увидеть, куда именно добавить этот код.

Имея возможность изменять точку зрения с течением времени, мы получаем дополнительные возможности сверх того, что разработали в главе 3. Теперь можем заняться отображением более интересной формы, чем октаэдр или сфера.

В.3. ЗАГРУЗКА И ОТОБРАЖЕНИЕ ЧАЙНИКА ИЗ ЮТЫ

В главе 2 мы вручную определили векторы, очерчивающие двухмерного динозавра, и точно так же могли бы вручную определить вершины любого трехмерного объекта, организовать их в тройки, представляющие треугольники, и построить поверхность в виде списка треугольников. Художники, создающие трехмерные модели, пользуются специализированными приложениями для позиционирования вершин в пространстве и последующего сохранения их в файлы. В этом разделе мы применим известную готовую трехмерную модель *чайник из Юты*. Отображение этого чайника — программа Hello World для программистов графики, простой и узнаваемый пример для тестирования.

Модель чайника хранится в файле `teapot.off` в примерах с исходным кодом, где расширение `.off` означает Object File Format (формат объектного файла). Это простой текстовый формат, определяющий многоугольники, составляющие поверхность трехмерного объекта, и трехмерные векторы, являющиеся

вершинами многоугольников. Файл `teapot.off` выглядит примерно так, как показано в листинге В.1.

Листинг В.1. Схема содержимого файла teapot.off

```
OFF
480  448  926
0  0  0.488037
0.00390625  0.0421881  0.476326
0.00390625  -0.0421881  0.476326
0.0107422  0  0.575333
...
4 324 306 304 317
4 306 283 281 304
4 283 248 246 281
...
```


В последних строках файла определяются грани. Первое число в каждой строке сообщает, каким многоугольником представлена грань. Цифра 3 обозначает треугольник, 4 — четырехугольник, 5 — пятиугольник и т. д. Большинство граней чайника представлены четырехугольниками. Следующие числа в каждой строке сообщают индексы вершин из предыдущих строк, которые образуют углы данного многоугольника.

В файле `teapot.py` в примерах исходного кода для приложения В вы найдете функции `load_vertices()` и `load_polygons()`, загружающие вершины и грани (многоугольники) из этого файла. Первая функция возвращает список из 480 векторов, определяющих вершины модели, вторая — список из 448 списков, каждый из которых содержит векторы вершин одной из 448 граней модели. Наконец, я написал третью функцию, `load_triangles()`, которая разбивает многоугольники с четырьмя и более вершинами, чтобы для построения модели использовать только треугольники.

Более глубокое изучение моего кода и попытки загрузить файл `teapot.off` я оставляю вам в качестве самостоятельного упражнения. А пока продолжим работу с треугольниками, загруженными в `teapot.py`, чтобы побыстрее нарисовать чайник и поэкспериментировать с ним. Другой шаг, который я пропущу, — определение функции для инициализации PyGame и OpenGL, чтобы не приходилось повторять ее каждый раз при рисовании модели. В `draw_model.py` вы найдете следующую функцию:

```
def draw_model(faces, color_map=blues, light=(1,2,3)):
...
```

Она принимает грани трехмерной модели (предполагается, что это правильно ориентированные треугольники), цветовую карту для моделирования

освещенности и вектор, определяющий местоположение источника света, и рисует модель. У нее есть также несколько именованных аргументов, представленных в главах 4 и 5. Так же как код рисования октаэдра, эта функция рисует любую переданную модель в цикле, снова и снова. В листинге В.2 показано, как я объединил упомянутые ранее функции в файле `draw_teapot.py`.

Листинг В.2. Загрузка треугольников граней чайника и передача их в вызов draw_model

```
from teapot import load_triangles
from draw_model import draw_model

draw_model(load_triangles())
```

В результате получается изображение чайника. Вы можете видеть круглую крышку, ручку слева и носик справа (рис. В.5).



Рис. В.5. Изображение чайника

Теперь, научившись отображать формы, более интересные, чем простая геометрическая фигура, можно и поэкспериментировать! Если вы уже прочитали главу 4, то узнали о математических преобразованиях, которые можно производить со всеми вершинами чайника для его перемещения и искажения в трехмерном пространстве. Далее приведу несколько упражнений для тех из вас, кто хотел бы изучить код отображения моделей под моим руководством.

В.4. УПРАЖНЕНИЯ

Упражнение В.1. Измените функцию `draw_model`, чтобы она позволяла отобразить входную фигуру с точки зрения, смещенной на любой угол. В частности, добавьте в `draw_model` именованный аргумент `glRotatefArgs` для передачи кортежа из четырех чисел, соответствующих четырем аргументам `glRotatef`, а затем добавьте в тело `draw_model` соответствующий вызов `glRotatef`, чтобы выполнить поворот.

Решение. Решение можно подсмотреть в файле `draw_model.py` в примерах исходного кода для книги, а пример использования — в `draw_teapot_glrotatef.py`.

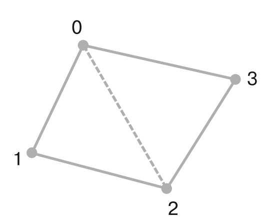
Упражнение В.2. Сколько секунд понадобится сцене, чтобы совершить полный оборот, если в каждом кадре вызывать `glRotatef(1,1,1,1)`?

Решение. Ответ зависит от частоты кадров. Вызов `glRotatef` выполняет поворот на 1° в каждом кадре. При частоте отображения 60 кадров в секунду скорость вращения составит 60° в секунду, и полный оборот на 360° будет выполнен за 6 с.

Упражнение В.3. Мини-проект. Реализуйте упомянутую ранее функцию `load_triangles()`, которая загружает содержимое файла `teapot.off` и создает список треугольников. Каждый треугольник должен задаваться тремя трехмерными векторами. Затем передайте свой результат в `draw_model()` и убедитесь, что получили то же самое изображение чайника.

Решение. В примерах исходного кода в файле `teapot.py` вы найдете готовую реализацию `load_triangles()`.

Подсказка. Четырехугольники можно превратить в пары треугольников, соединив их противоположные вершины.



Индексировав четыре вершины четырехугольника, можно получить два треугольника, образованные вершинами 0, 1, 2 и 0, 2, 3

Упражнение В.4. Мини-проект. Добавьте анимационный эффект в отображение чайника, изменяя значения аргументов в вызовах `gluPerspective` и `glTranslatef`. Это поможет вам понять влияние каждого из параметров.

Решение. В файле `animated_octahedron.py` в примерах исходного кода вы найдете реализацию поворота октаэдра на $360/5 = 72°$ в секунду путем обновления параметра угла `glRotatef` в каждом кадре. Можете попробовать выполнить похожие модификации либо с чайником, либо с октаэдром.

Пол Орланд

Математические алгоритмы для программистов. 3D-графика, машинное обучение и моделирование на Python

Перевел с английского А. Киселев

Руководитель дивизиона	*Ю. Сергиенко*
Ведущий редактор	*Н. Гринчик*
Литературные редакторы	*Н. Рощина*
Художественный редактор	*В. Мостипан*
Корректоры	*О. Андриевич, Н. Терех*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 01.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 03.11.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 60,630. Тираж 700. Заказ 0000.